New Primitives for Tackling Graph Problems and Their Applications in Parallel Computing

Peilin Zhong

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2021

Abstract

New Primitives for Tackling Graph Problems and Their Applications in Parallel Computing

Peilin Zhong

We study fundamental graph problems under parallel computing models. In particular, we consider two parallel computing models: Parallel Random Access Machine (PRAM) and Massively Parallel Computation (MPC). The PRAM model is a classic model of parallel computation. The efficiency of a PRAM algorithm is measured by its parallel time and the number of processors needed to achieve the parallel time. The MPC model is an abstraction of modern massive parallel computing systems such as MapReduce, Hadoop and Spark. The MPC model captures well coarse-grained computation on large data — data is distributed to processors, each of which has a sublinear (in the input data) amount of local memory and we alternate between rounds of computation and rounds of communication, where each machine can communicate an amount of data as large as the size of its memory. We usually desire fully scalable MPC algorithms, i.e., algorithms can work for any local memory size. The efficiency of a fully scalable MPC algorithm is measured by its parallel time and the total space usage (the local memory size times the number of machines).

Consider an n-vertex m-edge undirected graph G (either weighted or unweighted) with diameter D (the largest diameter of its connected components). Let $N = m + n$ denote the size of G. We present a series of efficient (randomized) parallel graph algorithms with theoretical guarantees. Several results are listed as follows:

- Fully scalable MPC algorithms for graph connectivity and spanning forest using $O(N)$ total space and $O(\log D \log \log_{N/n} n)$ parallel time.

- Fully scalable MPC algorithms for 2-edge and 2-vertex connectivity using $O(N)$ total space where 2-edge connectivity algorithm has $O(\log D \log \log_{N/n} n)$ parallel time, and 2-vertex connectivity algorithm has $O(\log D \cdot \log^2 \log_{N/n} n + \log D' \cdot \log \log_{N/n} n)$ parallel time. Here D' denotes the bi-diameter of G.

- PRAM algorithms for graph connectivity and spanning forest using $O(N)$ processors and $O(\log D \log \log_{N/n} n)$ parallel time.

- PRAM algorithms for $(1 + \epsilon)$-approximate shortest path and $(1 + \epsilon)$-approximate uncapacitated minimum cost flow using $O(N)$ processors and poly$(\log n)$ parallel time.

These algorithms are built on a series of new graph algorithmic primitives which may be of independent interests.

Table of Contents

List of Figures

Acknowledgements

My five years at Columbia University has been a wonderful journey in my life. During this journey, there are so many people I need to thank. But when I started to write this paragraph, I realized that it is impossible for me to express my full gratitude in a few pages of human languages.

First of all, I want to express my sincerest gratitude to my advisors, Alexandr Andoni, Clifford Stein and Mihalis Yannakakis. Their support for me covers many aspects, including not only research, but also writing/presenting skills, financial status, career planning, etc. Their ideas often provide me with various helpful guidance on research and life problems. I remember that I had several research projects stuck for a long time, and most of the problems are finally solved based on the ideas sparked from our countless meetings. I also remember that I faced many important choices in my academic career. The constructive suggestions they gave me gave me a clearer views of each direction and helped me make choices. Of course, during my Ph.D. studies, I sometimes feel stressed due to insufficient research progress or my poor writing/presentation. But my advisors never put any pressure on me. Instead, I can always be encouraged by them. In these five years, I also learned a lot from them. For research, they helped me find the research areas that I am passionate about, and I learned how to choose research topics and the way of thinking in the research process. For life, I learned how to plan my future better.

I also thank David P. Woodruff and Periklis A. Papakonstantinou. I was very fortunate to have taken Periklis's course *Algorithms and Models for Big Data* when I was an undergraduate student at Tsinghua University. It was my first time to explore interesting and elegant problems in theoretical computer science. I am even luckier that I had a chance to work on a course project that is supervised by David. After that project, I had several other research projects with David and I learned a lot of knowledge and techniques of math and theoretical computer science from him. Without their inspiration, I may not further pursue my PhD in theoretical computer science after my undergraduate study. It was also a wonderful experience when I was a visitor of David at IBM Almaden in summer 2018. That summer is filled with research discussions and meetings with other brilliant researchers. I had a great chance to explore diverse areas during that summer.

I would like to thank Omri Weinstein. I really enjoyed discussing research with him. He is always passionate about research and his passion motivates me a lot. I remember that we had a lot of meetings for research projects and he was always enthusiastic in sharing elegant results and ideas with me. For the areas that I am not familiar with, he was very patient to explain the background knowledges to me. He helped me dive into these areas deeply and made me learn a lot of interesting research problems, results and techniques.

During my PhD study, it was my great honor to intern at Google Research for two summers and one spring. I want to thank Alessandro Epasto, Hossein Esfandiari, Mohammad Mahdian, Vahab Mirrokni for hosting me. I also want to thank Ilya Razenshteyn for inviting me to visit Microsoft Research Redmond. All of my internship/visiting experiences were amazing.

Besides, I want to thank my other collaborators and friends AmirMohsen Ahanchi, Chang Xiao, Changxi Zheng, Chengyu Lin, Da Tang, Hengjie Zhang, Hongyang Zhang, Ji Xu, Lijie Chen, Lin F. Yang, Marina Knittel, MohammadTaghi HajiAghayi, Pengyu Chen, Robert E. Tarjan, Ruiqi Zhong, Ruosong Wang, S. Cliff Liu, Tianxiao Shen, Wei Hu, Ying Sheng, Yuchen Mo, Yuqing Ai, Zhao Song, Zhengyu Wang, among others, for the motivative discussions of research. I would especially to thank Changxi and Chang for a long term discussions of research in machine learning, Zhao, Ruosong and Lin for frequent constructive comments on my research, and my cousin Ruiqi for his long term helps in improving my academic writing. As a member of Columbia's theory group, I also want to thank everyone in the theory group. The time spent with theory group is full of pleasant memories.

I want to thank Xingming Wang, my coach when I was in the team of Olympiad in Informatics of my middle school, for inspiring my curiosity about algorithms.

Finally, with full of gratitude, I want to thank my father Wu Zhong, my mother Yinghua Liu, and my fiancée Minzi Mao for their love and encouraging support. Without their love and support, I would not be able to achieve what I have achieved.

To My Family

Chapter 1: Introduction

This thesis studies new *graph algorithms* which can be efficiently implemented in parallel computing settings and handle massive graph datasets. Many applications are modeled by large graphs. For example, social networks, transportation networks and the Internet can have more than a billion vertices. Therefore, it is challenging to efficiently compute results of interest (e.g., detect communities, create user recommendations, design trip routes, etc), and one way to handle these large graphs efficiently is to use multiple machines to process them in parallel.

Although many basic graph problems like connectivity, spanning tree and shortest path admit simple and efficient solutions under the sequential computing setting, it is surprisingly hard to take full advantage of large-scale parallelism and obtain a speedup proportional to the number of machines/processors. Therefore, an important research direction is to develop algorithms which fully utilize the power of large-scale parallelism for these fundamental graph problems.

Analyzing and developing new algorithms under parallel systems requires us to choose a theoretical model. Two popular models are the parallel random-access machine (PRAM) and the massively parallel computation (MPC) model [1, 2]. The PRAM model is a classic parallel model which has been studied for decades while the MPC model is a more accurate abstraction of today's parallel computing systems (such as MapReduce and Hadoop) and can capture better the computing power (e.g., the power of local computation on each machine) of these systems. It is known that any PRAM algorithm can be simulated in the MPC model. Therefore, PRAM algorithms are usually more general in the sense that they can be implemented in a wide range of parallel computing systems (such as a single machine with multiple processors and massively parallel computing systems), while the additional power of the MPC model allows us to design more efficient algorithms for massively parallel computing systems.

In this thesis, we will develop a line of new parallel graph algorithms under both the MPC

model and the PRAM model. Several main problems discussed in this thesis include *Graph Connectivity*, *Spanning Forest*, *Depth-First-Search Sequence*, *Biconnectivity*, 2-*Edge Connectivity*, *Shortest Path* and *Uncapacitated Minimum Cost Flow*.

1.1 MPC vs. PRAM

In recent years, several parallel systems, including MapReduce [3], Hadoop [4], Dryad [5], Spark [6], and others, have become successful in practice. This success has sparked a renewed interest in algorithmic ideas for these parallel systems.

An important theoretical direction has been to choose good models of these modern systems and design provably efficient parallel algorithms under these models. One choice is the PRAM model (see e.g., [7]), a classic parallel computing model which has been studied for several decades. In the PRAM model, there are multiple processors and a shared memory. The processors run synchronously. In each step, a processor can read a shared memory cell, do one unit of computation, and write a shared memory cell. The efficiency is measured by the number of parallel steps (parallel time) needed and the total number of operations made (work) over all processors. Since work is always upper bounded by the number of processors multiplied by the parallel time, sometimes we describe the efficiency of the PRAM algorithm by the number of processors and the parallel time required. The PRAM model characterizes the power of global parallel computation very well, and the PRAM algorithms are usually very general such that they can be implemented in different parallel computing settings, e.g., either on a single machine with multi-cores or in the massively parallel computing scenarios such as MapReduce. Although the PRAM model usually leads to general parallel algorithms, the drawback is that it cannot capture the power of local computation of a machine in massively parallel computing scenarios. To characterize the power of local computation, the work of [8, 1, 9, 2, 10] has led to the model of *Massively Parallel Computation* (MPC). The MPC model is a variant of the Bulk Synchronous Parallel (BSP) model [11]. We give a detailed description of the MPC model in Section 2.3. In particular, MPC allows N^{δ} space per machine (processor), where $\delta \in (0, 1)$ and N is the input size, with alternating rounds of

unlimited local computation, and communication of up to N^δ data per processor. At the end of the computation, the output is distributed on the output machines. An MPC algorithm can equivalently be seen as a small circuit, with arbitrary, N^δ-fan-in gates; the depth of the circuit is the parallel time. *Fully scalable* MPC algorithms, which work for any value δ, are usually desired because it is very common to be in the situation that we cannot enlarge the local memory of each machine (processor) when a larger input is given. Two important efficiency measures of a fully scalable MPC algorithm are parallel time and total space (the number of machines multiplied by the space of each machine).

Any PRAM algorithm can be simulated as a fully scalable algorithm on MPC in the same parallel time, and the total space needed is linear in the work of the PRAM algorithm [1, 9]. However, MPC is in fact more powerful than the PRAM: even computing the XOR of N bits requires near-logarithmic parallel-time on the most powerful CRCW PRAMs [12], whereas it takes constant, $O(1/\delta)$, parallel time on the MPC model. Thus, the two main algorithmic questions of this area are:

Question 1.1.1. *For which problems can we design fully scalable MPC algorithms that are* faster *than the best PRAM algorithms?*

Question 1.1.2. *For which problems can we design PRAM algorithms that are as fast as the best MPC algorithms? (For which problems, the additional power of the MPC model may not necessarily lead to faster algorithms?)*

For Question 1.1.1, we will give fully scalable MPC algorithms for Graph Connectivity, Spanning Forest, DFS Sequence, Biconnectivity and 2-Edge Connectivity that are faster than all known previous parallel algorithms (in both MPC and PRAM models).

For Question 1.1.2, we first show that our MPC graph connectivity and spanning forest algorithms can be extended to PRAM algorithms with the same parallel time. The number of processors needed is linear in the input size. This fact may tell us that the MPC model may not be strictly more powerful than the PRAM model on the connectivity and the spanning forest problems.

Then, we design PRAM algorithms for Shortest Paths and Uncapacitated Minimum Cost Flow that are faster than all known previous PRAM algorithms and fully scalable MPC algorithms, and are almost as fast as the previous fastest MPC algorithms (which are not fully scalable).

1.2 Problems, our results and comparison to prior results

In this section, we formally introduced the graph problems studied in this thesis. We will give a brief overview of our results and will compare our results to previous results. For all results shown in this section, the input graph G is either unweighted ($G = (V, E)$) or weighted ($G = (V, E, w)$ where $w : E \to \mathbb{Z}$ are weights of edges), with $n = |V|$ vertices, $m = |E|$ edges and size $N = |V| + |E|$. If G is an unweighted graph, we use D to be an upper bound of the diameter of any connected component of G.

We use (γ, δ)-MPC model to denote the MPC model where each machine has local space $\Theta(N^\delta)$ and the total space over all machines is $\Theta(N^{1+\gamma})$ (see the formal definition of (γ, δ)-MPC model in Section 2.3).

1.2.1 Graph connectivity

In the connectivity problem, the goal is to output the connected components of an input graph G, i.e. at the end of the computation, $\forall v \in V$, there is a unique tuple (v, y), where y is called the color of v. Any two vertices u, v have the same color if and only if they are in the same connected component.

The connectivity problem has been studied in the parallel literature for several decades. An $O(\log n)$ depth $\widetilde{O}(N)$[1] work PRAM algorithm has been known since [13]. Later, there is a line of PRAM algorithms on variants of PRAM models. The best known randomized PRAM connectivity algorithm would be [14] which has $O(\log n)$ depth and $O(N)$ work, and can be implemented in the weakest EREW PRAM model (see Section 2.2 for a detailed discussion of the EREW PRAM model). According to [1, 9], these algorithms can be simulated in the $(0, \delta)$-MPC model for arbi-

[1] $\widetilde{O}(f(n))$ denotes $f(n) \cdot \operatorname{poly}\log(f(n))$.

trary constant $\delta > 0$ in $O(\log n)$ rounds. A natural question is whether there is an MPC connectivity algorithm with fewer than $O(\log n)$ rounds. In [15], they show that there is an $O(1)$ rounds connectivity algorithm in the MPC model when the total space in the system is at least $n^{1+\Omega(1)}$. Later [16, 17] implied that there is an $O(1)$ rounds connectivity algorithm in the MPC model when the total space in the system is linear in the size of the graph. However, both algorithms require that the size of the local memory of a machine must be at least the number of vertices of the graph. This is a very restricted requirement since the number of vertices of a sparse graph is almost the size of the graph. Thus fully scalable algorithms would be more desirable in practice, i.e., we would like algorithms which work in the (γ, δ)-MPC model for any value of $\delta > 0$. We show a faster, fully scalable algorithm for the connectivity problem in the MPC model by parameterizing the parallel time as a function of the diameter D of the graph.

Theorem 1.2.1 (Connectivity in MPC, restatement of Theorem 4.4.4)**.** *For any $\gamma \in [0, 2]$ and any constant $\delta \in (0, 1)$, there is a randomized $(\gamma, \delta) -$ MPC algorithm which outputs the connected components of the graph G in $O(\min(\log D \cdot \log \frac{\log n}{\log(N^{1+\gamma}/n)}, \log n))$ expected parallel time.*

Notice that in the most restrictive case of $\gamma = 0$ and $m = n$, we obtain $O(\min(\log D \cdot \log\log n, \log n))$ time. When the total space is slightly larger, or the graph is slightly denser—i.e. $\gamma > c$ or $\log_n m > 1 + c$, where $c > 0$ is an arbitrarily small constant—then we obtain $O(\log D)$ time.

Surprisingly, by combining elegant hashing techniques with the new algorithmic ideas proposed by the algorithm mentioned in the above theorem, we also show a faster PRAM algorithm for the connectivity problem. In particular, we develop a near linear work randomized parallel algorithm in the COLLISION CRCW PRAM model in $O(\log(D) \cdot \log\log_{N/n} n)$ parallel time. In the COLLISION CRCW PRAM model, if multiple processors write the same shared memory cell at the same parallel step, then the value of the cell can be any corrupted value and the cell will be marked as collision. Notice that the COLLISION CRCW PRAM model is a weaker model than the ARBITRARY CRCW PRAM model. Therefore, our algorithm can be directly implemented in the ARBITRARY CRCW PRAM model. For more details of the COLLISION CRCW PRAM

model and the ARBITRARY CRCW PRAM model, we refer readers to Section 2.2.

Theorem 1.2.2 (Connecivity in PRAM, restatement of Theorem 4.6.48)**.** *There is a randomized COLLISION CRCW PRAM algorithm that outputs the connected components of the graph G with probability at least $1 - 1/\mathrm{poly}(N \log(n)/n)$ in $O(\log D \cdot \log\log_{N/n} n)$ depth using $O(N \cdot (\log D \cdot \log\log_{N/n} n))$ work.*

Remark 1.2.3. *Our MPC algorithm is later improved by [18]. They show that there is a randomized (γ, δ)-MPC algorithm solving connectivity problem in $O\left(\log D + \log\left(\frac{\log n}{N^{1+\gamma}/n}\right)\right)$ parallel time for any constant $\delta > 0$. By combining the algorithmic framework with hashing techniques, it can also be extended to a PRAM algorithm with $O(\log D + \log\log_{N/n} n)$ depth and $O(N \cdot (\log D + \log\log_{N^{1+\gamma}/n} n))$ work (see [19]).*

1.2.2 Spanning forest

A natural extension of the connectivity problem is the spanning forest (tree) problem. In the spanning forest problem, the goal is to output a subset of edges of an input graph G such that the output edges together with the vertices of G form a forest. For any two vertices, they are in the same tree in the forest if and only if they are in the same connected component in G. In the rooted spanning forest problem, in addition to the edges of the spanning forest, we are also required to orient the edge from child to parent, so that the parent-child pairs form a rooted spanning forest of the input graph G.

Connectivity algorithms in prior work can be naturally extended to solve the spanning forest problem. However, it is not obvious that our connectivity algorithm can also solve the spanning forest problem. To solve the spanning forest problem in the MPC model, additional steps are needed. We also give the following result for the spanning forest problem.

Theorem 1.2.4 (Spanning forest in MPC, restatement of Theorem 4.4.12)**.** *For any $\gamma \in [0, 2]$ and any constant $\delta \in (0, 1)$, there is a randomized $(\gamma, \delta) - \mathrm{MPC}$ algorithm which can output the rooted spanning forest for any graph $G = (V, E)$ in $O(\min(\log D \cdot \log \frac{\log n}{\log(N^{1+\gamma}/n)}, \log n))$ expected parallel*

time. Furthermore, the depth of the rooted spanning forest found is at most $D^{O\left(\log\left(\frac{\log n}{\log(N^{1+\gamma}/n)}\right)\right)}$.

Later, we use hashing techniques to show how to implement this idea in the PRAM model. We also give a faster PRAM algorithm for the spanning forest problem.

Theorem 1.2.5 (Spanning forest in PRAM, restatement of Theorem 4.6.30)**.** *There is a randomized ARBITRARY CRCW PRAM algorithm that outputs a spanning forest of the graph* G *with probability at least* $1 - 1/\text{poly}(N\log(n)/n)$ *in* $O(\log(D)\cdot\log\log_{N/n} n)$ *depth using* $O(N\cdot(\log(D)\cdot\log\log_{N/n} n))$ *work.*

We only show how to compute a spanning forest in the PRAM model and computing a rooted spanning forest remaining open.

1.2.3 Minimum spanning forest

In the minimum spanning forest problem, the goal is to compute a spanning forest of a weighted graph G such that the total weight of the forest is minimized. In the PRAM model, $O(\log n)$ depth $O(N)$ work algorithms are known (see e.g., [20] and references therein). In [15, 16, 17], they also give minimum spanning forest algorithms in the MPC model, which have $O(1)$ rounds. However, their algorithms still require that the local memory of a machine must be at least the number of vertices and thus are not fully scalable.

We show how to extend our fully scalable MPC connectivity algorithm to the fully scalable MPC (approximate) minimum spanning forest algorithm. The parallel time of our minimum spanning forest algorithms depends on D_{MSF} which is the hop diameter of a minimum spanning forest of G, i.e., the maximum number of edges (regardless of weights) on any simple path in the forest.

Theorem 1.2.6 (Minimum spanning forest, restatement of Theorem 4.5.3)**.** *Consider a weighted graph* G *with weights* $w : E \to \mathbb{Z}$ *such that* $\forall e \in E, |w(e)| \leq \text{poly}(n)$*. For any* $\gamma \in [0,2]$ *and any constant* $\delta \in (0,1)$*, there is a randomized* $(\gamma,\delta)-$MPC *algorithm which outputs a minimum spanning forest of* G *in* $O(\min(\log D_{MSF}\cdot\log(\frac{\log n}{1+\gamma\log n}),\log n)\cdot\frac{\log n}{1+\gamma\log n})$ *expected parallel time,*

where D_{MSF} is the diameter (with respect to the number of edges/hops) of a minimum spanning forest of G.

We note that we require the bounded weights condition merely to ensure that each weight is described by one word.

Theorem 1.2.7 (Approximate minimum spanning forest, restatement of Theorem 4.5.4)**.** *Consider a weighted graph G with weights $w : E \to \mathbb{Z}_{\geq 0}$ such that $\forall e \in E, |w(e)| \leq \text{poly}(n)$. For any $\epsilon \in (0,1)$, $\gamma \in [\beta, 2]$ and any constant $\delta \in (0,1)$, there is a randomized $(\gamma, \delta) - \text{MPC}$ algorithm which can output a $(1+\epsilon)$ approximate minimum spanning forest for G in $O(\min(\log D_{MSF} \cdot \log(\frac{\log n}{\log(N^{1+\gamma}/(\epsilon^{-1} n \log n))}), \log n))$ expected parallel time, where $\beta = \Theta(\log(\epsilon^{-1} \log n)/\log n)$, and D_{MSF} is the diameter (with respect to the number of edges/hops) of a minimum spanning forest of G.*

1.2.4 2-Edge connectivity

In the 2-edge connectivity problem, the goal is to output all bridges of the input graph G. A bridge in G is an edge whose removal increases the number of connected components of G. In the PRAM model, $O(\log n)$ depth $\widetilde{O}(N)$ work algorithms are known (see e.g., [21]). This PRAM algorithm implies a $(0, \delta)$-MPC algorithm with $O(\log n)$ parallel time for any constant $\delta \in (0,1)$ (by simulation [9]). There is an another MPC algorithm for 2-edge connectivity in $O(1)$ rounds [22]. However, the algorithm requires that the local memory of each machine is at least the number of vertices. Thus, the algorithm is not fully scalable.

We show a faster and fully scalable 2-edge connectivity algorithm in the (γ, δ)-MPC model for any constant $\delta > 0$ using the same parallel time as our connectivity algorithm.

Theorem 1.2.8 (2-Edge connectivity in MPC, restatement of Theorem 5.5.6)**.** *For any $\gamma \in [0,2]$ and any constant $\delta \in (0,1)$, there is a randomized (γ, δ)-MPC algorithm which outputs all the bridges of the graph G in $O\left(\log D \cdot \log \frac{\log n}{\log(N^{1+\gamma}/n)}\right)$ expected parallel time.*

1.2.5 Biconnectivity

Any two distinct edges e, e' of G are in the same biconnected component (block) of G if and only if there is a simple cycle which contains both e, e'. If we define a relation R such that eRe' if and only if $e = e'$ or e, e' are contained by a simple cycle, then R is a equivalence relation [23]. Thus, a biconnected component is an induced graph of an equivalence class of R. In the biconnectivity problem, the goal to find all biconnected components (blocks) of the input graph G. Since the biconnected components of G define a partition on E, we just need to color each edge, i.e., at the end of the computation, $\forall e \in E$, there is a unique tuple (e, c) stored on an output machine, where c is called the color of e, such that the edges e_1, e_2 are in the same biconnected components if and only if they have the same color. As the same as 2-edge connectivity, $O(\log n)$ depth $\widetilde{O}(N)$ work PRAM algorithms for biconnectivity are known (see e.g. [21]). [22] can also compute biconnected components in $O(1)$ MPC rounds. But as discussed previously, the algorithm requires the local memory of each machine to be at least the number of vertices, and thus the algorithm is not fully scalable.

We give faster, fully scalable algorithms for the the biconnectivity problem by parameterizing the parallel running time as a function of the diameter D and the bi-diameter D' of the graph G. The definition of bi-diameter is a natural generalization of the definition of diameter. If vertices u, v are in the same biconnected component, then the cycle length of (u, v) is defined as the minimum length of a simple cycle which contains both u and v. The bi-diameter D' of G is the largest cycle length over all the vertex pairs (u, v) where both u and v are in the same biconnected component.

Theorem 1.2.9 (Biconnectivity in MPC, restatement of Theorem 5.5.7)**.** *For any $\gamma \in [0, 2]$ and any constant $\delta \in (0, 1)$, there is a randomized (γ, δ)-MPC algorithm which outputs all the biconnected components of the graph G in $O\left(\log D \cdot \log^2 \frac{\log n}{\log(N^{1+\gamma}/n)} + \log D' \cdot \log \frac{\log n}{\log(N^{1+\gamma}/n)}\right)$ expected parallel time.*

The worst case is when the input graph is sparse and the total space available is linear in the input size, i.e., $N = n + m = O(n)$ and $\gamma = 0$. In this case, the parallel running time of our

algorithm is $O(\log D \cdot \log^2 \log n + \log D' \cdot \log \log n)$. If the graph is slightly denser ($m = n^{1+c}$ for some constant $c > 0$), or the total space is slightly larger ($\gamma > 0$ is a constant), then we obtain $O(\log D + \log D')$ time.

A cut vertex (articulation point) in the graph G is a vertex whose removal increases the number of connected components of G. Since a vertex v is a cut vertex if and only if there are two edges e_1, e_2 which share the endpoint v and e_1, e_2 are not in the same biconnected component, our algorithm can also find all the cut vertices of G.

Furthermore, we show that our biconnectivity algorithm can be further extended to find an open ear decomposition. Suppose G is a biconnected graph, i.e., G has only one biconnected component. Then an open ear decomposition is a partition $E_1, E_2, \cdots, E_s$ of the edge set E such that E_1 is a set of edges of a simple cycle, and each of $E_2, E_3, \cdots, E_s$ is a set of edges of a simple path. Furthermore, the internal vertices on the path of E_i do not appear in $E_1, E_2, \cdots, E_{i-1}$, and the two end vertices of the path of E_i are distinct.

Theorem 1.2.10 (Open ear decomposition in MPC, restatement of Theorem 5.7.6)**.** *For any $\gamma \in [0,2]$ and any constant $\delta \in (0,1)$, there is a randomized (γ, δ)-MPC algorithm which outputs an open ear decomposition of G in $O\left(\log D \cdot \log^2 \frac{\log n}{\log(N^{1+\gamma}/n)} + \log D' \cdot \log \frac{\log n}{\log(N^{1+\gamma}/n)}\right)$ expected parallel time.*

1.2.6 Shortest path and uncapacitated minimum cost flow

Uncapacitated minimum cost flow and shortest path are two more difficult problems than connectivity since any feasible s-t flow or any s-t path is a certificate that s and t are in the same connected component. Even in the classic PRAM model, previous algorithms for these problems are far from attaining the efficiency we would like to spend. In the uncapacitated minimum cost flow problem, each vertex $u \in V$ has a demand $b(u) \in \mathbb{R}$ satisfying $\sum_{u \in V} b(u) = 0$, the goal is to determine the flow $f(x,y)$ for each edge $\{x,y\} \in E$ such that $f(x,y) = -f(y,x)$, $\forall u \in V, \sum_{\{x,u\} \in E} f(x,u) = b(u)$, and the total flow cost $\sum_{\{x,y\} \in E} |f(x,y)| \cdot w(x,y)$ is minimized. Single source shortest path (SSSP) is one of the most fundamental problems in computer science,

and it is also a special case of uncapacitated minimum cost flow. Given a source vertex $s \in V$, if we set demand $b(u) \geq 0$ for all $u \neq s \in V$ and $b(s) = -\sum_{u \neq s \in V} b(u)$, then the optimal cost of the flow is exactly $\sum_{u \in V \setminus \{s\}} b(u) \cdot \text{dist}_G(s, u)$. Thus, the flow routes on the shortest path tree is the optimal flow. Standard sequential single source shortest path algorithms with (nearly) optimal running time have been known for several decades [24, 25, 26]. In contrast, parallelizing these algorithms has been a challenge. The exact shortest paths can be computed by the standard path-doubling (Floyd-Warshall) algorithm in poly$(\log n)$ depth using $O(n^3)$ total work, for an n-node m-edge graph. This result has been improved in a long line of work [27, 28, 29, 30, 31, 32, 33]. Nevertheless, the state-of-the-art algorithms have either $\Omega(n^{2.1})$ work or $\Omega(n^{0.1})$ depth. In order to achieve algorithms with better bounds on work and depth, researchers have turned to approximation algorithms. Building on the idea of hopsets [34], a series of papers, including [35, 34, 36, 37, 38] give $(1 + \epsilon)$-approximation algorithms. Yet again, every prior algorithm with m poly$(\log n)$ work has at least $\Omega(n^\rho)$ depth, and the ones with poly$(\log n)$ depth do $\Omega(mn^\rho)$ work, where $\rho > 0$ is an arbitrary small constant. In particular, none of the prior algorithms achieve poly$(\log n)$ depth and m poly$(\log n)$ work simultaneously. In fact, there was no known parallel algorithm with poly$(\log n)$ parallel time and m poly$(\log n)$ work that approximates the shortest path even up to a poly$(\log n)$ factor. We develop a parallel $(1 + \epsilon)$-approximate shortest path algorithm with poly$(\log n)$ depth and m poly$(\log n)$ work.

Theorem 1.2.11 (Parallel $(1 + \epsilon)$-approximate single source shortest path tree and potentials, restatement of Theorem 6.4.46)**.** *Given a graph $G = (V, E, w)$, a vertex s and an error parameter $\epsilon \in (0, 0.5)$, there is a PRAM algorithm which outputs an approximate shortest path tree T and distance labels $\{d(u) \mid u \in V\}$ in ϵ^{-2} poly log(n) parallel time using expected $\widetilde{O}(\epsilon^{-3}m)$ work. With probability at least* 0.99*, the following properties hold:*

1. $\forall u \in V, \text{dist}_T(s, u) \leq (1 + \epsilon)\,\text{dist}_G(s, u)$.
2. $\forall u \in V, d(u) \geq (1 - \epsilon)\,\text{dist}_G(s, u)$.
3. $\forall u, v \in V, |d(u) - d(v)| \leq \text{dist}_G(u, v)$.

Although we present our parallel algorithms in the PRAM model, they can also be implemented in the MPC model. By applying the simulation methods [1, 9], our PRAM algorithm can be directly simulated in MPC. The obtained MPC algorithm has $\text{poly}(\log n)$ rounds and only needs $m \cdot \text{poly}(\log n)$ total space. Furthermore, it is also fully scalable, i.e., the memory size per machine can be allowed to be m^{δ} for any constant $\delta \in (0, 1)$. To the best of our knowledge, this is the first MPC algorithm which computes $(1 + \epsilon)$-approximate shortest path using $\text{poly}(\log n)$ rounds and $m\,\text{poly}(\log n)$ total space when the memory of each machine is upper bounded by $n^{1-\Omega(1)}$. Previous work on shortest path in the MPC model include [39] when the memory size per machine is $o(n)$, and simulations of shortest path algorithms from the Congested Clique model [40, 41, 42, 43, 44, 45] when the memory size per machine is $\Omega(n)$ [17].

As part of obtaining the parallel SSSP algorithm, we also develop a parallel for the uncapacitated min-cost flow problem.

Theorem 1.2.12 (Parallel uncapacitated minimum cost flow, restatement of Theorem 6.4.13)**.** *Given a graph $G = (V, E, w)$, a demand vector $b \in \mathbb{R}^n$ and an error parameter $\epsilon \in (0, 0.5)$, there is a PRAM algorithm which outputs a $(1 + \epsilon)$-approximate solution to the uncapacitated minimum cost flow problem with probability at least 0.99 in $\epsilon^{-2}\,\text{poly}\log(n)$ parallel time using $\widetilde{O}(\epsilon^{-2}m)$ expected work.*

The prior state-of-the-art sequential algorithm solves the uncapacitated minimum cost flow problem in $m \cdot 2^{O(\sqrt{\log n})}$ time [46]. Hence, our parallel uncapacitated min-cost flow algorithm also improves this previously best-known running time. In the MPC model, [43] implies an algorithm using $O(\log(n))$ rounds and $\widetilde{O}(N)$ total space in the system. However, it requires that the local memory size of each machine to be at least the number of vertices and thus it is not fully scalable. In contrast, by applying the simulation methods [1, 9], our PRAM algorithm can be directly simulated in MPC. The obtained MPC algorithm has $\text{poly}(\log n)$ rounds and only needs $m \cdot \text{poly}(\log n)$ total space. It is also fully scalable.

1.2.7 Hardness results

Although we mainly focus on the design of algorithms, we also show some hardness results.

Conditional hardness for directed reachability. We consider the reachability question in directed graphs, for which we show similar to the above results are unlikely. In particular, we show that if there is a fully scalable multi-query directed reachability $(0, \delta)$ – MPC algorithm with $n^{o(1)}$ parallel time and polynomial local running time, then we can compute Boolean Matrix Multiplication (see Section 7.1 for a formal definition of the problem) in $n^{2+\epsilon+o(1)}$ time for arbitrarily small constant $\epsilon > 0$. We note that the equivalent problem for *undirected graphs* can be solved in $O(\log D \log\log n)$ parallel time via Theorem 4.4.4.

Theorem 1.2.13 (Directed Reachability vs. Boolean Matrix Multiplication, restatement of Theorem 7.1.1)**.** *If there is a polynomial local running time fully scalable* (γ, δ) – MPC *algorithm which can answer* $|V| + |E|$ *pairs of reachability queries simultaneously for any directed graph* $G = (V, E)$ *in* $O(|V|^{\alpha})$ *parallel time, then there is a sequential algorithm which can compute the multiplication of two* $n \times n$ *Boolean matrices in* $O(n^2 \cdot n^{2\gamma+\alpha+\epsilon})$ *time, where* $\epsilon > 0$ *is a constant which can be arbitrarily small.*

Conditional hardness for biconnectivity. A conjectured hardness for the connectivity problem is the *one cycle vs. two cycles* conjecture: for any $\gamma \geq 0$ and any constant $\delta \in (0, 1)$, any (γ, δ)-MPC algorithm requires $\Omega(\log n)$ parallel time to determine whether the input n-vertex graph is a single cycle or contains two disjoint length $n/2$ cycles. This conjectured hardness result is widely used in the MPC literature [1, 2, 47, 48, 49]. Under this conjecture, we show that $\Omega(\log D')$ parallel time is necessary for the biconnectivity problem, even when $D = O(1)$, i.e., the diameter of the graph is a constant.

Theorem 1.2.14 (Hardness of biconnectivity in MPC, restatement of Theorem 7.3.2)**.** *For any* $\gamma \geq 0$ *and any constant* $\delta \in (0, 1)$*, unless there is a* (γ, δ)*-MPC algorithm which can distinguish the following two instances: 1) a single cycle with n vertices, 2) two disjoint cycles each contains*

$n/2$ vertices, in $o(\log n)$ parallel time, any (γ, δ)-MPC algorithm requires $\Omega(\log D')$ parallel time for testing whether a graph G with a constant diameter is biconnected.

Unconditional hardness for connectivity in CREW PRAM. We consider the connectivity problem in the Concurrent Read Exclusive Write PRAM model, i.e., the event that a shared memory cell is accessed by two processors at the same time is not allowed. By reduction from the OR of n bits, we show following two lower bounds for connectivity in CREW PRAM model.

Theorem 1.2.15 (Restatement of Theorem 7.5.3)**.** *Any deterministic CREW PRAM algorithm which solves connectivity for an n-vertex graph with diameter at most 2 needs $\Omega(\log n)$ parallel time even when the number of processors and the number of shared memory cells are unlimited.*

Theorem 1.2.16 (Restatement of Theorem 7.5.4)**.** *Any randomized CREW PRAM algorithm which uses $O(n)$ processors to solve connectivity for an n-vertex graph with diameter 2 with successful probability at least $2/3$ needs $\Omega(\log \log n)$ parallel time.*

1.3 New primitives for tackling graph problems

In this section, we briefly introduce the new techniques which are used to obtain the results shown in the previous section.

1.3.1 Truncated broadcasting and double-exponential speed problem size reduction

Two main techniques developed for our connectivity and spanning forest are *truncated broadcasting* (see Section 3.1) and *double-exponential speed problem size reduction* (see Section 3.2). Later in Section 4.6, we show how to use a hashing technique to implement truncated broadcasting.

Double exponential speed problem size reduction: This is a general technique which solves a problem in $O(\log \log n)$ iterations using small space. To be more precise, for any problem characterized by a size parameter n, if there is a subroutine which uses total space $\Theta(m)$ to reduce the problem size such that the reduced problem size is n/k for $k = (m/n)^{\Omega(1)}$, then we can solve the problem in $O(m)$ total space by iteratively calling the subroutine $O(\log \log n)$ times. The proof is

sketched as follows. When the problem size is $O(1)$, we can solve the problem easily. Let n_i be the problem size after the i-th iteration of calling the subroutine. Suppose the subroutine uses $\Theta(m)$ space to reduce the problem size to at most n_i/k_i for $k_i = (m/n_i)^c, c = \Omega(1)$. Then, after repeating the subroutine i times, the problem size is $n_i \leq n_{i-1}/(m/n_{i-1})^c \leq n \cdot (n/m)^{(1+c)^i-1}$. Thus, after $O(\log_{1+c} \log_{m/n} n)$ iterations, the problem size will be reduced to $O(1)$.

Truncated broadcasting: This technique is a rediscovery of a technique proposed by [31]. We show a procedure to compute a set of close vertices for each vertex using small total space budget and small number of iterations. More precisely, given a parameter b, we can use $\widetilde{O}(|E| + |V| \cdot b^2)$ total space and i rounds to compute either all vertices within 2^i hops for each vertex or all b-closest vertices for each vertex. The high level idea is by truncated doubling (broadcasting). At the beginning each vertex maintains a list containing its direct neighbors. If the size of the list is already at least b, then it only stores b arbitrary vertices in the list. In each iteration, each vertex v expends its list by adding vertices from the list of u into the list of v for each u in the list of v. If the size of the list of v is larger than b, then it only keeps b vertices in the list and drops other vertices. A simplified description of the algorithm is shown below:

1. For each vertex $v \in V$, initialize a list $L^{(0)}(v)$ containing b arbitrary neighbors (including v itself) of v. For $u \in L^{(0)}(v)$, initialize $\text{dist}^{(1)}(v, u) \leftarrow 1$. Let t be the number of iterations.

2. For $i = 1 \rightarrow t$:

   (a) For any two vertices v, u, (conceptually) initialize $\text{dist}^{(2^i)}(v, u) \leftarrow \infty$.

   (b) Consider each vertex v. For each vertex $x \in L^{(i-1)}(v)$ and each vertex $u \in L^{(i-1)}(x)$, if $\text{dist}^{(2^{i-1})}(v, x) + \text{dist}^{(2^{i-1})}(x, u) < \text{dist}^{(2^i)}(v, u)$, update $\text{dist}^{(2^i)}(v, u) \leftarrow \text{dist}^{(2^{i-1})}(v, x) + \text{dist}^{(2^{i-1})}(x, u)$.

   (c) For each vertex v, add u to list $L^{(i)}(v)$ if $\text{dist}^{(2^i)}(v, u)$ is one of the b smallest values among $\text{dist}^{(2^i)}(v, x)$ for $x \in V$. If there is a tie, take the vertex with a smaller label.

3. Output $L^{(t)}(v)$ for each vertex v and output $\text{dist}^{(2^t)}(v, u)$ for each $u \in L^{(t)}(v)$.

Hashing based truncated broadcasting: We show that truncated broadcasting procedure can be simply implemented using random hashing techniques. For a given parameter b, we can use $\widetilde{O}(|E| + |V| \cdot b^c)$ total space and i rounds to compute either all vertices within 2^i hops for each vertex or all b-closest vertices for each vertex, where $c > 2$ is a constant. Although we have a blow-up in the exponent of b, it is sufficient to given many efficient PRAM algorithms.

1.3.2 Recursive DFS sequence construction via leaf sampling

The DFS sequence of a tree is a variant of the Euler tour representation of the tree. It is a crucial building block in the 2-edge connectivity algorithm and the biconnectivity algorithm. For an n-vertex tree T, [50] gives an $O(\log n)$ parallel time PRAM algorithm for the Euler tour representation of T. However, since their construction method will destroy the tree structure, it is hard to get a faster MPC algorithm based on this framework.

We give a new framework for constructing DFS sequence (see Section 5.3 and Section 5.2 for more details). First of all, we compute a rooted tree, reducing the problem to computing a DFS sequence for a rooted tree. The idea is motivated by TeraSort [51]. If the size of the tree is small enough, we can easily generate its DFS sequence. Otherwise, our algorithm can be roughly described as follows.

1. Sample leaves $l_1, l_2, \cdots, l_s$ uniformly at random.
2. Determine the order of sampled leaves in the DFS sequence.
3. Compute the DFS sequence $\widetilde{A}$ of the tree which only consists of sampled leaves and their ancestors.
4. Recursively compute the DFS sequence A_v of every root-v subtree which does not contain any sampled leaf.
5. Merge $\widetilde{A}$ and all the A_v.

Notice that the number of leaves in each subtree can be at most n/s in the fourth step. The number of levels of the recursion can be much smaller than $\log(n)$ if we set $s = \omega(1)$.

Theorem 1.3.1 (Restatement of Theorem 5.3.9)**.** *For any $\gamma \in [0,2]$ and any constant $\delta \in (0,1)$, there is a randomized $(\gamma,\delta)-$* MPC *algorithm (Algorithm 19) which can output a Depth-First-Search sequence for any tree graph $G = (V,E)$ in $O(\min(\log D \cdot \log(1/\gamma'), \log n))$ parallel time, where $n = |V|$, D is the diameter of G, and $\gamma' = (1+\gamma)\log_n \frac{2N}{n^{1/(1+\gamma)}}$. The success probability is at least* 0.98*. In addition, if the algorithm fails, then it will return* FAIL*.*

1.3.3 New tools for shortest path and comparison to prior approaches

To obtain our shortest path and uncapacitated minimum cost flow algorithms, we develop new tools, which we present next and which may be of independent interest. It is most natural to present these results in the context of two related approaches to parallel shortest path algorithms — hopsets and continuous optimization techniques.

We note that some of our results have new consequences beyond parallel algorithms, including faster sequential algorithms and constructions where none were previously known.

Prior approach — hopsets: One iteration of Bellman-Ford can be implemented efficiently in parallel, and therefore, for graphs in which an approximate shortest path has a small number of hops (edges) we already have an efficient algorithm. Motivated by this insight, researchers have proposed adding edges to a graph in order to make an approximate shortest path with a small number of edges between every pair of vertices. Formally, for a given graph $G = (V,E,w)$ with weights $w : E \to \mathbb{R}_{\geq 0}$, a hopset is an edge-set H with weights $w_H : H \to \mathbb{R}_{\geq 0}$. Let $\widetilde{G}$ be the union graph $(V, E \cup H, w \cup w_H)$. We define $\text{dist}^{(h)}_{\widetilde{G}}(u,v)$, the h-hop distance in $\widetilde{G}$, to be the length of the shortest path between $u, v \in V$ which uses at most h hops (edges) in $\widetilde{G}$. Then H is an (h,ϵ)-*hopset* of G if $\forall u, v \in V$, $\text{dist}^{(h)}_{\widetilde{G}}(u,v)$ is always a $(1+\epsilon)$-approximation to the shortest distance between u and v in the graph G. There is a three-way trade-off between h, ϵ, and $|H|$, which was studied in [34, 52, 37, 38, 53], leading to some of the aforementioned algorithms.

Surprisingly, a hard barrier arose: [54] showed that the size of H must be $\Omega(n^{1+\rho})$ for any $h \leq \text{poly}(\log n)$, $\epsilon < \frac{1}{\log n}$ and some constant $\rho > 0$. Thus, it is impossible to directly apply hopsets to compute a $(1+\epsilon)$-approximate shortest path in $\text{poly}(\frac{\log n}{\epsilon})$ parallel time using $m\,\text{poly}(\frac{\log n}{\epsilon})$ work

for sparse graphs G, when $|E| = O(n)$.

Low Hop Emulator: To bypass this hardness, we introduce a new notion — *low hop emulator* — which has a weaker approximation guarantee than hopsets, but has stronger guarantees in other ways (see Section 6.2 for more details). A low hop emulator $G' = (V, E', w')$ of G is a sparse graph with $n \operatorname{poly}(\log n)$ edges satisfying two properties. First, the distance between every pair of vertices in G' is a $\operatorname{poly}(\log n)$ approximation to the distance in G. The second property is that G' has a low hop diameter, i.e., a shortest path between every pair of two vertices in G' only contains $O(\log \log n)$ number of hops (edges).

Another notion related to low hop emulator is spanner. Although spanner can also preserve pairwise distances approximately, there are two major differences between a spanner and a low hop emulator. Firstly, the edge set of a spanner is a subset of edges of the original graph while the edges of a low hop emulator may not be in the original graph. Secondly, the number of edges on some shortest paths in a spanner may be much larger than $\operatorname{poly}(\log n)$ while the number of edges on any shortest path in a low hop emulator can be at most $O(\log \log n)$.

We give an efficient parallel sparse low hop emulator construction algorithm. To the best of our knowledge, it was not even clear whether sparse low hop emulators exist, and thus no previous algorithm was known even in the sequential setting.

Theorem 1.3.2 (Low hop emulator, restatement of Theorem 6.4.3)**.** *For any $k \geq 1$, any graph G admits a low hop emulator G', with expected size of $\widetilde{O}\left(n^{1+\frac{1}{k}}\right)$,satisfying:*

$$\forall u, v \in V, \operatorname{dist}_G(u, v) \leq \operatorname{dist}_{G'}(u, v) \leq \operatorname{poly}(k) \cdot \operatorname{dist}_G(u, v),$$

and with the hop diameter at most $O(\log k)$. Furthermore, there is a PRAM algorithm computing the emulator G' in $\operatorname{poly}\log(n)$ *parallel time using $\widetilde{O}(m + n^{1+\frac{2}{k}})$ expected work.*

Notice that, setting $k = \log n$, we can compute a low hop emulator with expected size $\widetilde{O}(n)$ and hop diameter $O(\log \log n)$ in $\operatorname{poly}\log(n)$ parallel time using $\widetilde{O}(m)$ expected work. The approximation ratio in this case is $\operatorname{poly}\log(n)$.

We now highlight two main features that make a low hop emulator stronger than hopsets. Firstly, the low hop emulator can be computed in $\text{poly}(\log n)$ parallel time using $m\,\text{poly}(\log n)$ work while the same guarantees cannot be simultaneously achieved by hopsets. Secondly, the $O(\log\log n)$-hop distances in low hop emulator G' satisfy the *triangle inequality* while the h-hop distances in the union graph $\widetilde{G}$ of original graph G and the (h,ϵ)-hopset do not.

An immediate application of the first feature is a $\text{poly}(\log n)$-approximate single source shortest path (SSSP) algorithm in $\text{poly}(\log n)$ parallel time using $m\,\text{poly}(\log n)$ work (Corollary 6.4.5). We remark that when we use the hop-distance to approximate the exact distance in G, we only need to use the edges from the low hop emulator while we also need to use original edges if we use hopsets.

The second feature is crucial for designing parallel algorithms for Bourgain's embedding [55] (Corollary 6.4.6), metric tree embedding [56, 57] (Corollary 6.4.8) and low diameter decomposition [58] (Corollary 6.4.7), using $\text{poly}(\log n)$ depth and $m\,\text{poly}(\log n)$ work. [57] introduced a notion similar to low hop emulators, and it also has the second feature mentioned above. In contrast, their emulator graph is a complete graph, and the construction is based on $\left(\text{poly}(\log n), \frac{1}{\text{poly}(\log n)}\right)$-hopsets.

Continuous optimization via compressible preconditioner: To boost the approximation ratio of shortest paths from $\text{poly}(\log n)$ to $(1+\epsilon)$, we employ continuous optimization techniques. Recently, continuous optimization techniques have been successfully applied to design new efficient algorithms for many classic combinatorial graph problems, e.g., [59, 60, 61, 62, 63, 64, 65, 46, 66, 67]. Most of them can be seen as "boosting" a coarse approximation algorithm to a more accurate approximation algorithm. Oftentimes, to fit into a general optimization framework, the "coarse" approximation must be for a more general problem — in our case, for the uncapacitated minimum cost flow, also known as the transshipment problem. Following this approach, the work of [43] develops near-optimal uncapacitated min-cost flow algorithms in the distributed and streaming settings based on the gradient descent algorithm. Their algorithm can be seen as boosting a $\text{poly}(\log n)$ approximate solver for the uncapacitated min-cost flow problem to a $(1+\epsilon)$ approximate solver,

but with one crucial difference: it requires a poly($\log n$) approximate solver for the *dual problem*. Hence it is not clear how to leverage their algorithm for our goal as the aforementioned techniques do not seem applicable to the dual of uncapacitated min-cost flow.

We develop an algorithm for the uncapacitated min-cost flow problem by opening up Sherman's framework [46] and combining it with new techniques. There is a fundamental challenge in adopting Sherman's framework, beyond implementing it in the parallel setting. Sherman's original algorithm solves the uncapacitated minimum cost flow problem in $m \cdot 2^{O(\sqrt{\log n})}$ sequential time. Hence, if we obtain a parallel uncapacitated min-cost flow algorithm with m poly($\log n$) total work, we cannot avoid improving this best-known running time of $m \cdot 2^{O(\sqrt{\log n})}$ to m poly($\log n$).

To handle the challenge mentioned above, we develop a novel *compressible* preconditioner (see Section 6.3.2 and Section 6.3.3 for more details). By using our compressible preconditioner inside Sherman's framework, we improve the running time of $(1+\epsilon)$-approximate uncapacitated min-cost flow from $m \cdot 2^{O(\sqrt{\log n})}$ to m poly($\log n$). Furthermore, we show that such a compressible preconditioner can be computed in poly($\log n$) parallel time using m poly($\log n$) work. This preconditioner relies crucially on our low hop emulator ideas. Thus, we can solve $(1+\epsilon)$-approximate uncapacitated minimum cost flow in poly($\log n$) parallel time using m poly($\log n$) work.

Recursive path/tree construction: While the above techniques are sufficient for estimating the *value* of the shortest path, one additional challenge arises when we want to compute a $(1+\epsilon)$-approximate shortest *path*. In particular, the continuous optimization framework produces an approximate shortest path *flow*, which is not necessary integral and, more crucially, may contain cycles. We address this challenge by developing a novel recursive algorithm (see Section 6.4.5 and Section 6.4.6 for more details) based on random walks, and which uses a coupling argument.

By combining our random walk based algorithm with the dual solution of the uncapacitated minimum cost flow obtained by Sherman's algorithm [66], we can use the framework developed by [43] to further compute single source shortest paths.

1.4 Summary of techniques and algorithms

We summarize our new graph algorithmic primitives and the resulting parallel graph algorithms. Figure 1.1 sketches the dependencies between our new primitives and the parallel graph algorithms.

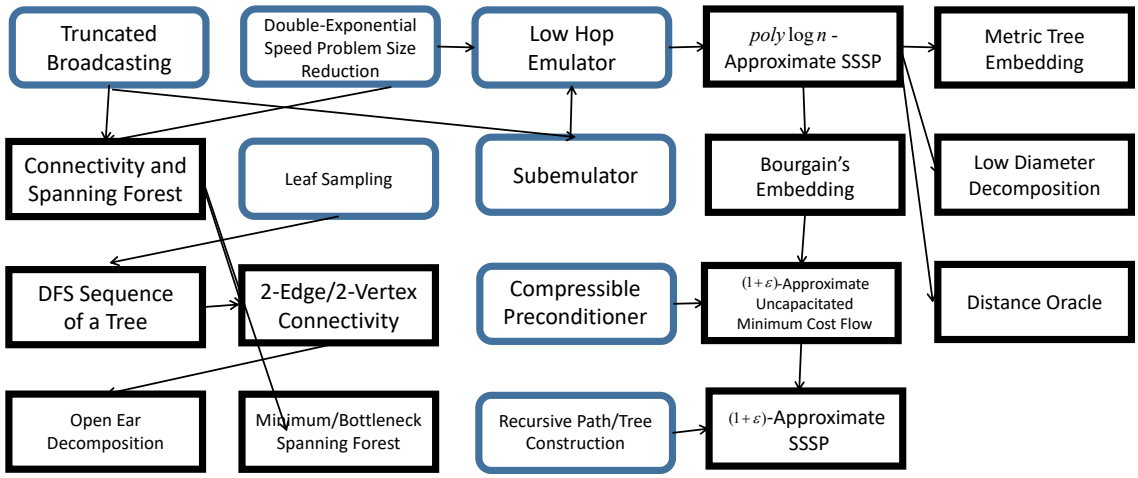

Figure 1.1: A summary of new primitives and parallel graph algorithms. Blue rounded rectangles indicate new primitives. Black rectangles indicate results of parallel graph algorithms.

In the following, we give corresponding sections and theorems of algorithms shown in Figure 1.1:

- **Truncated Broadcasting:** Section 3.1.
- **Double-Exponential Speed Problem Size Reduction:** Section 3.2.
- **Low Hop Emulator:** Theorem 6.4.3.
- poly$(\log n)$**-Approximate Single Source Shortest Paths (SSSP):** Corollary 6.4.5.
- **Metric Tree Embedding:** Corollary 6.4.8.

- **Connectivity and Spanning Forest:** Theorem 4.4.4, Theorem 4.4.12, Theorem 4.6.48, Theorem 4.6.30.
- **Leaf Sampling:** Lemma 5.2.18, Lemma 5.3.6.
- **Subemulator:** Theorem 6.4.1.
- **Bourgain's Embedding:** Corollary 6.4.6.
- **Low Diameter Decomposition:** Corollary 6.4.7.
- **DFS Sequence of a Tree:** Theorem 5.3.9.
- 2**-Edge/**2**-Vertex Connectivity:** Theorem 5.5.6, Theorem 5.5.7.
- **Compressible Preconditioner:** Section 6.3.2, Section 6.4.4.
- $(1+\epsilon)$**-Approximate Uncapacitated Minimum Cost Flow:** Theorem 6.4.13.
- **Distance Oracle:** Theorem 6.4.4.
- **Open Ear Decomposition:** Theorem 5.7.6.
- **Minimum/Bottleneck Spanning Forest:** Section 4.5.
- **Recursive Path/Tree Construction:** Section 6.4.5, Section 6.4.6.
- $(1+\epsilon)$**-Approximate Single Source Shortest Paths (SSSP):** Theorem 6.4.46.

1.5 Related papers

This thesis is related to several published papers. The connectivity and spanning forest algorithms in Chapter 4 are related to [68, 19]. The DFS sequence, biconnectivity and 2-edge connectivity algorithms in Chapter 5 are related to [68, 69]. The shortest path and uncapacitated minimum cost flow algorithms in Chapter 6 are related to [70]. [68] was in FOCS 2018. [69] was in ICALP 2019. [19] was in SPAA 2020. [70] was in STOC 2020.

The results shown in Section 4.6.5, Section 5.6, Section 6.4.6 and Section 7.5 were not previously public.

Chapter 2: Preliminaries and Parallel Computing Models

2.1 Notation

We use $[n]$ to denote the set $\{1, 2, \cdots, n\}$. For a set V, 2^V denotes the family of all the subsets of V, i.e., $2^V = \{S \mid S \subseteq V\}$. We use $\log(\cdot)$ to denote $\log_2(\cdot)$ and use $\ln(\cdot)$ to denote $\log_e(\cdot)$. We use $\widetilde{O}(f(n))$ to denote $O(f(n) \cdot \log(f(n)))$. We use $\text{poly}(f(n))$ to denote $f(n)^{O(1)}$. For $a, b \geq 0, \alpha \geq 1$, we say a is an α-approximate to b if $b \leq a \leq \alpha \cdot b$.

We use $G = (V, E)$ to denote an unweighted undirected graph with vertex set V and edge set E. We use $G = (V, E, w)$ to denote a weighted undirected graph with vertex set V, edge set E and weight $w(e) = w(u, v) = w(v, u) = w(\{v, u\})$ for each edge $e = \{u, v\} \in E$. If there are multiple edges between u and v, we take the edge with minimum edge weight $w(u, v)$. For $v \in V$, let $w(v, v)$ be 0. For two vertices $u, v \in V$, if the label of u is smaller than the label of v, then we denote it as $u < v$. In some situation, we need to regard an unweighted graph G as a weighted graph. If not otherwise specified, each edge e in G has weight $w(e) = 1$. Thus, all definitions for weighted graphs also apply for unweighted graphs. We will only consider graphs with non-negative weights. Consider a tuple $p = (u_0, u_1, u_2, \cdots, u_h) \in V^{h+1}$. If $\forall i \in [h]$, either $u_i = u_{i-1}$ or $\{u_{i-1}, u_i\} \in E$, then p is a path between u_0 and u_h. The number of hops of p is h, and the length of p is defined as $w(p) = \sum_{i=1}^{h} w(u_{i-1}, u_i)$. For an unweighted graph, the length of a path is equal to the number of hops of the path. For $u, v \in V$, let $\text{dist}_G(u, v)$ denote the length of the shortest path between u, v, i.e., $\text{dist}_G(u, v) = w(p^*)$, where the path p^* between u, v satisfies that $\forall$path p between u, v, $w(p^*) \leq w(p)$. If there is no path between u and v, then $\text{dist}_G(u, v) = \infty$. Similarly, $\text{dist}_G^{(h)}(u, v)$ denotes the h-hop distance between u, v, i.e., $\text{dist}_G^{(h)}(u, v) = w(p')$, where the h-hop path p' between u, v satisfies that $\forall h$-hop path p between u, v, $w(p') \leq w(p)$. The diameter of G is defined as $\max_{u,v \in V: \text{dist}_G(u,v) < \infty} \text{dist}_G(u, v)$. The hop diameter of G is defined as

the minimum value of $h \in \mathbb{Z}_{\geq 0}$ such that $\forall u, v \in V, \mathrm{dist}_G(u,v) = \mathrm{dist}_G^{(h)}(u,v)$. We use $\mathrm{diam}(G)$ to denote the hop diameter of G. Notice that the diameter and the hop diameter are equal if G is an unweighted graph. For $S \subseteq V, v \in V$, we define $\mathrm{dist}_G(v,S) = \mathrm{dist}_G(S,v) = \min_{u\in S} \mathrm{dist}(u,v)$. Similarly, we define $\mathrm{dist}_G^{(h)}(v,S) = \mathrm{dist}_G^{(h)}(S,v) = \min_{u\in S} \mathrm{dist}_G^{(h)}(u,v)$. If G is clear in the context, we use $\mathrm{dist}(\cdot,\cdot)$ and $\mathrm{dist}^{(h)}(\cdot,\cdot)$ for short. If u, v are not in the same connected component in G, then $\mathrm{dist}_G(u,v) = \infty$. If u, v are in the same connected component in G, then $\mathrm{dist}_G(u,v) < \infty$. For $v \in V$, $\{u \in V \mid \mathrm{dist}_G(u,v) < \infty\}$ is the set of all the vertices in the same connected component as v. $(v_1, v_2, \cdots, v_k) \in V^k$ is a cycle of length $k-1$ if $v_1 = v_k$ and $\forall i \in [k-1], \{v_i, v_{i+1}\} \in E$. We say a cycle $(v_1, v_2, \cdots, v_k)$ is simple if $k \geq 4$ and each vertex only appears once in the cycle except v_1 (v_k). Consider two different vertices $u, v \in V$. We use $\mathrm{cyclen}_G(u,v)$ to denote the minimum number of edges of a simple cycle which contains both vertices u and v. If there is no simple cycle which contains both u and v, $\mathrm{cyclen}_G(u,v) = \infty$. $\mathrm{cyclen}_G(u,u)$ is defined as 0. The hop bi-diameter of G, $\text{bi-diam}(G)$, is defined as $\max_{u,v\in V:\mathrm{cyclen}_G(u,v)\neq\infty} \mathrm{cyclen}_G(u,v)$. For undirected graph, we also call the hop bi-diameter as bi-diameter.

Consider two weighted graphs $G = (V, E, w)$ and $G' = (V, E', w')$. If $\forall u, v \in V, \mathrm{dist}_G(u,v) \leq \mathrm{dist}_{G'}(u,v) \leq \alpha \cdot \mathrm{dist}_G(u,v)$ for some $\alpha \geq 1$, then G' is called an α-emulator of G.

For $v \in V, \Gamma_G(v)$ denotes the set of neighbors of v in G, i.e. $\Gamma_G(v) = \{u \in V \mid \{v,u\} \in E\} \cup \{v\}$. Notice that we always regard v as a neighbor of v itself. For $b \in [|V|]$ and vertices $u, v \in V$, if $|\{x \in \Gamma_G(v) \mid \mathrm{dist}_G(x,v) < \mathrm{dist}_G(u,v) \vee (\mathrm{dist}_G(x,v) = \mathrm{dist}_G(u,v) \wedge x < v)\}| < b$, then we say u is a b-closest neighbor (or a b-closest direct neighbor) of v in G. In particular, for any $b \geq 1$, v is always a b-closest neighbor of v itself. We define $\Gamma_{G,b}(v)$ as the set of all b-closest neighbors of v in G.

Given $r \in \mathbb{Z}_{\geq 0}$, for $v \in V$, we define $\mathrm{Ball}_G(v,r) = \{u \in V \mid \mathrm{dist}_G(u,v) \leq r\}$, and $\mathrm{Ball}_G^\circ(v,r) = \{u \in V \mid \mathrm{dist}_G(u,v) < r\}$. Given $b \in [|V|]$, for $v \in V$, let $r_{G,b}(v)$ satisfy that $|\mathrm{Ball}_G(v, r_{G,b}(v))| \geq b$ and $|\mathrm{Ball}_G^\circ(v, r_{G,b}(v))| < b$. For two vertices $u, v \in V$, if u satisfies that $\mathrm{dist}_G(u,v) < \infty$ and $|\{x \in V \mid \mathrm{dist}_G(x,v) < \mathrm{dist}_G(u,v) \vee (\mathrm{dist}_G(x,v) = \mathrm{dist}_G(u,v) \wedge x < v)\}| < b$, then we say u is a b-closest vertex of v in G. Similarly, if u satisfies that $\mathrm{dist}_G^{(h)}(u,v) < \infty$ and $|\{x \in V \mid \mathrm{dist}_G^{(h)}(x,v) <$

$\mathrm{dist}_G^{(h)}(u,v) \vee (\mathrm{dist}_G^{(h)}(x,v) = \mathrm{dist}_G^{(h)}(u,v) \wedge x < v)\}| < b$, then we say u is a b-closest vertex of v under h-hop distance in G. In particular, for any $b \geq 1$, v is always a b-closest vertex of v itself. We define $\mathrm{Ball}_{G,b}(v)$ as the set of all b-closest vertices to v in G. If there is no ambiguity, we just use $\mathrm{Ball}(v,r)$, $\mathrm{Ball}^\circ(v,r)$, $r_b(v)$, $\mathrm{Ball}_b(v)$, to denote $\mathrm{Ball}_G(v,r)$, $\mathrm{Ball}_G^\circ(v,r)$, $r_{G,b}(v)$, $\mathrm{Ball}_{G,b}(v)$ respectively for short.

For a vector $x \in \mathbb{R}^m$ we use $\|x\|_1$ to denote the ℓ_1 norm of x, i.e., $\|x\|_1 = \sum_{i=1}^m |x_i|$. We use $\|x\|_\infty$ to denote the ℓ_∞ norm of x, i.e., $\|x\|_\infty = \max_{i\in[m]} |x_i|$. Given a matrix $A \in \mathbb{R}^{n\times m}$, we use A_i, A^j and $A_{j,i}$ to denote the i-th column, the j-th row and the entry in the i-th column and the j-th row of A respectively. We use $\|A\|_{1\to 1}$ to denote the operator ℓ_1 norm of A, i.e., $\|A\|_{1\to 1} = \sup_{x:x\neq 0} \frac{\|Ax\|_1}{\|x\|_1}$. A well-known fact is that $\|A\|_{1\to 1} = \max_{i\in[m]} \|A_i\|_1$. We use $\mathbf{1}_n$ to denote an n dimensional all-one vector and we use $\mathbf{0}_n$ to denote an n dimensional all-zero vector. We use $\mathrm{sgn}(a)$ to denote the sign of a, i.e., $\mathrm{sgn}(a) = 1$ if $a \geq 0$, and $\mathrm{sgn}(a) = -1$ otherwise. We use $\mathrm{nnz}(\cdot)$ to denote the number of non-zero entries of a matrix or a vector.

2.2 The PRAM models

A parallel random-access machine (PRAM) is a shared-memory abstract machine. It consists of a set of processors, each of which has a small private memory, and a large shared memory. Each memory cell contains $O(\log n)$ bits where n is the size of the input. The processors run synchronously. In one step, a processor can read a cell in shared memory, write to a cell in shared memory, or do a constant amount of local computation. Based on different behaviors of concurrently reading/writing the same shared memory cell, PRAM model has multiple variants (see e.g. [7]): exclusive read exclusive write (EREW) PRAM, concurrent read exclusive write (CREW) PRAM and concurrent read concurrent write (CRCW) PRAM. More precisely, a shared memory cell can be read by only one processor at a time in the exclusive read PRAM model while it can be read by multiple processors at a time in the concurrent read PRAM model. Similarly, a shared memory cell can be written by only one processor at a time in the exclusive write PRAM model while it can be written by multiple processors at a time in the concurrent write PRAM model. In

the CRCW PRAM model, when multiple processors write a shared memory cell at the same time, there are different strategies to determine the final value written in the cell. Consider the case that multiple processors write the same shared memory cell at a time. In the COMMON CRCW PRAM model, the processors must write the same value. In the COLLISION CRCW PRAM model [71], the cell can be an arbitrary corrupted value and will be marked as corrupted. In the ARBITRARY CRCW PRAM model, the cell can be an arbitrary written value. In the PRIORITY CRCW PRAM model, only the processor with the highest priority can successfully write the value. In the COMBINING CRCW PRAM model, the final value in the cell is a function of all written values. Some common choices of the combing functions in the COMBINING CRCW PRAM model are MIN, MAX, SUM and etc. It is easy to see that the later CRCW PRAM models mentioned above are stronger than the earlier CRCW PRAM models, i.e., COMBINING CRCW PRAM model is the strongest CRCW PRAM model while COMMON CRCW PRAM model is the weakest. Two standard measures of the efficiency of a parallel algorithm in the PRAM model are work (total time over the processors) and depth (parallel time). According to [72, 73], if we allow poly($\log n$) blow-up in the depth and work, we are able to simulate a specific PRAM algorithm in another variant of the PRAM model. We refer readers to [74] for a survey of basic algorithms in PRAM models.

2.3 The MPC model

In this section, let us introduce the Massively Parallel Computation (MPC) model. Unlike the PRAM model which has been studied for several decades, the MPC model was formally proposed in the last decade [8, 1, 9, 2, 10]. Although every PRAM algorithm can be simulated in the MPC model with the same parallel time [1, 9], many basic operations which are easy in PRAM become non-trivial in the MPC model. We will describe the details of how to implement several basic operations under the MPC model in this section later.

In the MPC model, we have p machines indexed from 1 to p each with memory size s words, where n is the number of words of the input and $p \cdot s = O(n^{1+\gamma}), s = \Theta(n^{\delta})$. Here $\delta \in (0, 1)$ is a constant, $\gamma \in \mathbb{R}_{\geq 0}$, and a word has $\Theta(\log(s \cdot p))$ bits. Thus, the total space in the system is only

$O(n^\gamma)$ factor more than the input size n, and each machine has local memory size sublinear in n. When $0 \leq \gamma \leq O(1/\log n)$, the total space is just linear in the input size. The computation proceeds in rounds. At the beginning of the computation, the input is distributed on the local memory of $\Theta(n/s)$ input machines. Input machines and other machines are identical except that input machine can hold a part of the input in its local memory at the beginning of the computation while each of other machines initially holds nothing. In each round, each machine performs computation on the data in its local memory, and sends messages to other machines (including the sender itself when it wants to keep the data) at the end of the round. Although any two machines can communicate directly in any round, the total size of messages (including the self-sent messages) sent or received of a machine in a round is bounded by s, its local memory size. In the next round, each machine only holds the received messages in its local memory. At the end of the computation, the output is distributed on the output machines. Output machines and other machines are identical except that output machine can hold a part of the output in its local memory at the end of the computation while each of other machines holds nothing. We call the above model (γ, δ) − MPC model. The model is exactly the same as the model MPC(ϵ) defined by [2] with $\epsilon = \gamma/(1 + \gamma - \delta)$ and the number of machines $p = O(n^{1+\gamma-\delta})$. Since we care more about the space used by the algorithm, we use (γ, δ) to characterize the model, while in [2] they use parameter ϵ to characterize the repetition of the data. The main time complexity measure here is the number of rounds R required to solve the problem.

2.3.1 Basic MPC operations

Sorting. One of the most important algorithms in MPC model is sorting. The following theorem shows that there is an efficient sorting algorithm in the MPC model.

Theorem 2.3.1 ([9, 75])**.** *Sorting can be solved in c/δ rounds in* $(0, \delta)$ − MPC *model for any constant $\delta \in (0, 1)$, where $c \geq 0$ is a universal constant. Precisely, there is an algorithm $\mathcal{A}$ in* $(0, \delta)$ − MPC *model such that for any set S of n comparable items stored $O(n^\delta)$ per machine on input machines, $\mathcal{A}$ can run in c/δ rounds and leave the n items sorted on the output machines, i.e.*

the ouput machine with smaller index holds a smaller part of $O(n^\delta)$ items.

Notice that for any $\delta' \geq \delta$, $O(1)$ number of machines with $\Theta(n^{\delta'})$ memory can always simulate the computation of $O(n^{\delta'-\delta})$ number of machines with $\Theta(n^\delta)$ memory. Thus, if an algorithm $\mathcal{A}$ can solve a problem in (γ, δ) − MPC model in $R(n)$ rounds, then $\mathcal{A}$ can be simulated in (γ', δ') − MPC model still in $R(n)$ rounds with all $\gamma' \geq \gamma, \delta' \geq \delta$.

Indexing. In the indexing problem, a set $S = \{x_1, x_2, \cdots, x_n\}$ of n items are stored $O(n^\delta)$ per machine on input machines. The output is

$$S' = \{(x, y) \mid x \in S, y - 1 \text{ is the number of items before } x\}$$

of n pairs stored $O(n^\delta)$ per machine on output machines. Here, "an item $x' \in S$ is before $x \in S$" means that x' is held by a input machine with a smaller index, or x', x are stored in the same input machine but x' has a smaller local memory address.

Prefix sum. In the prefix sum problem, a set $S = \{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$ of n (item, number) pairs are stored $O(n^\delta)$ per machine on input machines. The output is

$$S' = \left\{ (x, y') \;\middle|\; (x, y) \in S, y' - y = \sum_{(\widetilde{x},\widetilde{y}) \text{ is before } (x,y)} \widetilde{y} \right\}$$

of n pairs stored $O(n^\delta)$ per machine on output machines. Here, "an pair $(\widetilde{x}, \widetilde{y}) \in S$ is before $(x, y) \in S$" means that $(\widetilde{x}, \widetilde{y})$ is held by an input machine with a smaller index, or $(\widetilde{x}, \widetilde{y}), (x, y)$ are stored in the same input machine but $(\widetilde{x}, \widetilde{y})$ has a smaller local memory address. Notice that indexing problem is a special case of prefix sum problem.

Theorem 2.3.2 ([9])**.** *Indexing/prefix sum problem can be solved in c/δ rounds in $(0, \delta)$ − MPC model for any constant $\delta \in (0, 1)$, where $c \geq 0$ is a universal constant.*

Once each item has an index, it is able to reallocate them onto the machines.

Load balance. Sometimes, local computations of a machine may generate new data. When some machines are not able to keep the new data generated, we need to do loading balance. Fortunately, this operation can be done in constant number of rounds of computations.

For an arbitrary constant $\delta \in (0,1)$, we are able to spend constant number of rounds to reallocate the data in $(0,\delta)-$MPC model such that if a machine is not empty, the size of its local data is at least n^{δ}/k and is at most $2n^{\delta}/k$ where $k > 1$ is an arbitrary constant. The method is very simple, we can use the algorithm mentioned in Theorem 2.3.2 to index each data item, and then send them to the corresponding machine.

Predecessor. In the predecessor problem, a set $S = \{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$ of n (item, $0/1$) pairs are stored $O(n^{\delta})$ per machine on input machines. The output machines are all input machines. If an input (also output) machine holds a tuple $(x_i, y_i) \in S$ at the beginning of the computation, then at the end of the computation, that machine should still hold the tuple (x_i, y_i). In addition, if an input (also output) machine holds a tuple $(x, 0) \in S$ at the beginning of the computation, then at the end of the computation, that machine should hold a tuple (x, x') such that $(x', 1) \in S$, and $(x', 1)$ is the last tuple occurred before $(x, 0)$. Here, "$(x', 1)$ is before $(x, 0)$" means that $(x', 1)$ is held by an input machine with a smaller index, or $(x', 1), (x, 0)$ are stored in the same input machine but $(x', 1)$ has a smaller local memory address.

Theorem 2.3.3 ([9])**.** *Predecessor problem can be solved in c/δ rounds in $(0,\delta)-$*MPC *model for any constant $\delta \in (0,1)$, where $c \geq 0$ is a universal constant.*

Roughly speaking the algorithm is as the following: firstly, build a $\Theta(n^{\delta})$ branching tree on the machines, then follows by bottom-up stages to collect the last $(x_l, 1)$ tuple in each large interval and then follows by top-down stages to compute the predecessors of every prefix. For completeness, we describe the algorithm for predecessor problem in the following:

Predecessor Algorithm:

- **Setups:**

- There are $2p = \Theta(n^\delta)$ machines indexed from 1 to $2p$ each with local memory size $s = \Theta(n^\delta)$. The machine with index from $p + 1$ to $2p$ are input/output machines.
- $(x_1, y_1), \cdots (x_n, y_n)$ are stored on input/output machine $p + 1$ to $2p$, where $\forall i \in [n], y_i \in \{0, 1\}$.
- The goal: If an input machine holds a tuple (x, y) with $y = 0$, then it will create a tuple (x, x') at the end of the computation, where (x', y') is the last tuple with $y' = 1$ stored before (x, y).

- **Bottom-up stage ($O(1/\delta)$ constant rounds):**
  - Let $d = s/10$ be the branching factor.
  - In the i^{th} round, each machine j with j in the range $\lfloor p/d^{i-1} \rfloor + 1$ to $\lfloor (2p-1)/d^{i-1} \rfloor + 1$ sends the last (x_l, y_l) tuple with $y_l = 1$ in its local memory to machine $\lfloor (j-1)/d \rfloor + 1$. If machine j does not have any tuple with $y_l = 1$, it just sends an arbitrary tuple to machine $\lfloor (j-1)/d \rfloor + 1$.
  - Until the end of the computation, machine j sends itself messages to keep the data. The stage ends when machine 1 receives messages.

- **Top-down stage ($O(1/\delta)$ constant rounds):**
  - Let $d = s/10$ be the branching factor.
  - In the i^{th} round, each machine j with j in the range $\lfloor d^{i-2} \rfloor + 1$ to $\min(d^{i-1}, p)$ sends to each machine h in the range $(j-1)d + 1$ to $\min(j \cdot d, 2p)$ a tuple (x_l, y_l) which is the last tuple with $y_l = 1$ appeared before machine h.
  - The stage ends when machine $2p$ receives messages.

- **The last round:**
  - Machine $i \in \{p + 1, \cdots, 2p\}$ scans its local memory, for each tuple (x, y) with $y = 0$, create a tuple (x, x') where (x', y') is the last tuple stored before (x, y) with $y' = 1$.

2.3.2 Data organization

In this section, we introduce the method to organize the data in the system.

Set. Let $S = \{x_1, x_2, \cdots, x_m\}$ be a set of m items, and each item x_i can be described by $O(1)$ number of words. If $x \in S$ is equivalent to that there is a unique machine which holds a pair ("S", x) in its local memory, then we say that S is stored in the system. Here "S" is the name of the set S and can be described by $O(1)$ number of words.

Let $\mathcal{S} = \{S_1, S_2, \cdots, S_m\}$ be a set of m sets, where $\forall i \in [m]$, S_i is stored in the system, and the name "S_i" of each set S_i can be described by $O(1)$ number of words. If $S \in \mathcal{S}$ is equivalent to that there is a unique machine which holds a pair ("$\mathcal{S}$", "S") in its local memory, then we say $\mathcal{S}$ is stored in the system. Here "$\mathcal{S}$" is the name of $\mathcal{S}$ and can be described by $O(1)$ number of words.

Let S be a set stored in the system. If machine i has a pair ("S", x), then we say that the element x of S is held by the machine i. If every element of S is held by a machine with index in $\{i, i+1, \cdots, j\}$, then we say S is stored on the machine i to the machine j.

The total space needed to store S is $\Theta(m)$.

Mapping. Let $f : U \to H$ be a mapping from a finite set U to a set H. In the following, we show how to use a set to represent a mapping.

Definition 2.3.4 (Set representation of a mapping)**.** *Let $f : U \to H$ be a mapping from a finite set U to a set H. Let $S = \{(x, y) \mid x \in U, y = f(x)\}$. then the set S is a set representation of the mapping f.*

Let U be a finite set where each element of U can be described by $O(1)$ number of words. Let S be a set representation of the mapping $f : U \to H$. If S is stored in the system, then we say f is stored in the system. If S is stored on the machine i to the machine j, then f is stored on the machine i to the machine j. At any time of the system, there can be at most one set representation S of f stored in the system. Furthermore, the name of S is "f" which is the same as the name of mapping f, and can be described by $O(1)$ number of words.

The total space needed to store f is the total space needed to store S, and thus is $\Theta(|U|)$.

Sequence. Let $A = (a_1, a_2, \cdots, a_m)$ be a sequence of m elements. In the following, we show how to use a set to represent a sequence.

Definition 2.3.5 (Set representation of a sequence)**.** *Let $A = (a_1, a_2, \cdots, a_m)$ be a sequence of n elements. If a set $S = \{(x_1, y_1), (x_2, y_2), \cdots, (x_m, y_m)\} \subseteq \mathbb{R} \times \{a_1, a_2, \cdots, a_m\}$ satisfies $x_1 < x_2 < \cdots < x_m, y_1 = a_1, y_2 = a_2, \cdots, y_m = a_m$, then the set S is a set representation of the sequence A. Furthermore, if $x_1 = 1, x_2 = 2, \cdots, x_m = m$, then S is a standard set representation of A.*

Let A be a sequence of elements where each element can be described by $O(1)$ number of words. Let S be a set representation of the sequence A. If S is stored in the system, then we say A is stored in the system. If S is stored on the machine i to the machine j, then A is stored on the machine i to the machine j. At any time of the system, there can be at most one set representation S of A stored in the system. Furthermore, the name of S is "A" which is the same as the name of sequence A, and can be described by $O(1)$ number of words.

The total space needed to store A is the total space needed to store S, and thus is $\Theta(m)$.

2.3.3 Set operations

In this section, we introduce some MPC model operations for sets.

Duplicates removing. There are n tuples stored in the machines. But there are some duplicates of them. The goal is to remove all the duplicates. To achieve this, we can just sort all the tuples. After sorting, if a tuple is different from its previous tuple, then we keep it. Otherwise, we remove the tuple.

Sizes of sets. Suppose we have k sets $S_1, S_2, \cdots, S_k$ stored in the system. Our goal is to get the sizes of all the sets. We can firstly sort all the tuples such that the tuples from the same set are consecutive. Then we can calculate the index of each tuple. Every machine can scan all the tuples in its local memory. If x is an element of set S_i and has the smallest/largest index y, then create a pair ("boundary of 'S_i' ", y). Next, we sort all the created pairs. For each set S_i, there are two

pairs ("boundary of 'S_i' ", y_1), ("boundary of 'S_i' ", y_2) stored on the same machine. For each pair of tuples ("boundary of 'S_i' ", y_1), ("boundary of 'S_i' ", y_2) with $y_1 < y_2$, the machine can generate a new tuple ("f", ("S_i", $y_2 - y_1 + 1$)). Finally, there will be a mapping f stored in the system, where $f(S_i) = |S_i|$. The total number of rounds is a constant.

Copies of sets. Suppose we have k sets $S_1, S_2, \cdots, S_k$ stored in the system. Let $s_1, s_2, \cdots, s_k \in \mathbb{Z}_{\geq 1}$. Suppose if a machine holds an element $x \in S_i$, the machine also knows the value s_i. The goal is to create sets $S_{1,1}, S_{1,2}, \cdots, S_{1,s_1}, S_{2,1}, S_{2,2}, \cdots, S_{2,s_2}, \cdots, S_{k,s_k}$ and make them stored in the system, where $S_{i,j}$ is a copy of S_i.

The idea is very simple: for an element $x \in S_i$, we need to make s_i copies ("$S_{i,1}$", x), ("$S_{i,2}$", x), $\cdots$, ("S_{i,s_i}", x) of tuple ("S_i", x). But the issue is that s_i may be very large such that it is not able to generate all the copies of a tuple on a single machine. For the above reason, we implement it in three steps: firstly we compute the new "position" of each original tuple among all the copies, then send the original tuples to their new "positions", and finally filling the gap by generating copies between any two adjacent original tuples. Precisely, each machine can scan its local memory, and assign each tuple ("S_i", x) a weight s_i. Then we can use prefix sum algorithm (See Theorem 2.3.2) to compute the prefix sum of each tuple ("S_i", x). The prefix sum pos("S_i", x) of a tuple ("S_i", x) denotes the new "position" of the last copy of this tuple when all the copies are generated. Let $n = \sum_{i=1}^{k} s_i \cdot |S_i|$. Let machine 1 to t be t empty machines each maintains $s/10$ "positions", i.e. machine 1 has "positions" 1 to $s/10$, machine 2 has "positions" $s/10 + 1$ to $2s/10$, and so on. Let $t \cdot s/10 = \Theta(n)$. The machine which holds tuple ("S_i", x) sends the tuple ("S_i", x) to the "position" pos("S_i", x) $- s_i + 1$, and sends the tuple ("S_{i,s_i}", x) to the "position" pos("S_i", x). Then each machine $i \in [t]$ scans its "positions". If a "position" received a tuple, the machine marks that "position" as "1". Otherwise, the machine marks that position as "0". Now we can apply the predecessor algorithm (See Theorem 2.3.3) such that each empty "position" learns its predecessor tuple. If the predecessor tuple of an empty "position" l is ("S_i", x), and the predecessor tuple is at "position" l', then create a tuple ("$S_{i,l-l'}$", x) at this empty position. Thus, at the end of all the computations,

$S_{1,1}, S_{1,2}, \cdots, S_{1,s_1}, S_{2,1}, S_{2,2}, \cdots, S_{2,s_2}, \cdots, S_{k,s_k}$ are stored on the system.

Indexing elements in sets. Suppose we have k sets $S_1, S_2, \cdots, S_k$ stored in the system. The goal is to compute a mapping f such that $\forall i \in [k], x \in S_i$, x is the $f(S_i, x)^{\text{th}}$ element of S_i.

To achieve this goal, we can sort (See Theorem 2.3.1) all the tuples such that the elements from the same set are stored consecutively on several machines. Then we can run indexing algorithm (See Theorem 2.3.2) to compute the global index of each tuple. Next, each machine scans its local data. If ("S_i", x) is in the local memory, and x is the first element of S_i, then the machine marks this tuple as "1". For other tuples in the local memory, the machine marks them as "0". Then we can invoke predecessor algorithm (See Theorem 2.3.3) on all the tuples. At the end of the computation, each machine scans its all tuples. For a tuple ("S_i", x) with global index l, the machine determine the index of x in S_i based on the global index l' of its predecessor ("S_i", x). Precisely, the machine creates a tuple ("f", (("S_i", x), $l - l' + 1$)) stored in the memory. Thus at the end of the computation, the desired mapping f is stored in the system.

Set merging. Suppose we need to merge several sets $S_1, S_2, \cdots, S_k$ stored on the system, i.e. create a new set $S = \bigcup_{i=1}^{k} S_i$. To implement this operation, each machine scans its local memory. If there is a tuple ("S_i", x) in its memory, then it creates a tuple ("S", x). Finally, we just need to remove all the duplicates.

Set membership. Suppose we have k sets $S_1, S_2, \cdots, S_k$ stored in the system. There is an another set $Q = \{(x_1, y_1), \cdots, (x_q, y_q)\}$ also stored in the system where x_i is the name of a set S, and y_i is an item. The goal is to answer whether y_i is in S.

To achieve this, we can firstly sort all the tuples. For tuple with form ("S_i", x), the first key is S_i, the second key is x, and the third key is $-\infty$ which has the highest priority. For tuple with form ("Q", (x, y)), the first key is x, the second key is y, and the third key is ∞ which has the lowest priority. The comparison in the sorting procedure firstly compare the first key, then the second key, and finally the third key. After sorting, for each tuple with form ("S_i", x), we mark it as "1". For

each tuple with form ("Q", (x, y)), we mark it as "0". Now we can apply the predecessor algorithm (See Theorem 2.3.3). For each tuple ("Q", (x, y)), if its predecessor is ("S", y) where x is the name of "S", then we create a tuple ("f", $((x, y), 1)$); Otherwise, we create a tuple ("f", $((x, y), 0)$). Thus, at the end of the computation, there is a mapping f stored on the system such that for each $(x, y) \in Q$, if x is the name of some set S_i, and $y \in S_i$, then $f(x, y) = 1$; Otherwise $f(x, y) = 0$.

2.3.4 Mapping operations

In this section, we introduce some MPC model operations for mapping. The most important operation is called Multiple queries. It can be used to simulate the simultaneous access of memory cells in a single parallel step in PRAM.

Multiple queries. We have k sets $S_1, S_2, \cdots, S_k$ stored in the system. Without loss of generality, $S_1, S_2, \cdots, S_t$ $(t \leq k)$ are sets representations of mappings (See Definition 2.3.4) $f_1 : U_1 \to H_1, f_2 : U_2 \to H_2, \cdots, f_t : U_t \to H_t$ respectively. When a machine does local computation, it may need to query some values which are in the form $f_i(u)$ for some $u \in U_i$. The following lemma shows that we can answer all the such queries simultaneously in constant number of rounds in $(0, \delta)$ − MPC model for all constant $\delta \in (0, 1)$. It means that we can use constant number of rounds to simulate concurrent read operations on a shared memory where $S_1, \cdots, S_k$ are stored in the shared memory.

Lemma 2.3.6 (Multiple queries)**.** *Let $\delta \in (0, 1)$ be an arbitrary constant. There is a constant number of rounds algorithm $\mathcal{A}$ in $(0, \delta)-$*MPC *model which satisfies the following properties. The input of $\mathcal{A}$ contains two parts. The first part are k sets $S_1, S_2, \cdots, S_k$ stored (See Section 2.3.2 for data organization of sets) on the input machines, where $S_1, S_2, \cdots, S_t$ $(t \leq k)$ are sets representations of mappings (See Definition 2.3.4) $f_1 : U_1 \to H_1, f_2 : U_2 \to H_2, \cdots, f_t : U_t \to H_t$ respectively. The second part is a set $Q = \{(x_1, y_1, z_1), (x_2, y_2, z_2), \cdots, (x_q, y_q, z_q)\}$ stored on the input machines, where $\forall (x, y, z) \in Q$, x is the name "f_i" of the mapping f_i for some $i \in [t]$, y is an element in U_i, and z is the index of the input machine which holds the element (x, y, z) of Q. The total input size $n = |Q| + \sum_{i=1}^{k} |S_i|$. The output machines are all the input machines. $\forall i \in [k], x \in S_i$, if the*

element x of S_i is held by the input (also output) machine j, then at the end of the computation, the element x of S_i should still be held by the output (also input) machine j. Let Q' be the set $\{(x, y, z, w) \mid \exists (x, y, z) \in Q, w = f_i(y), \text{where } x \text{ is the name of } f_i\}$. At the end of the computation, Q' is stored on the output (also input) machines such that $\forall (x, y, z, w) \in Q'$, the element (x, y, z, w) of Q' is held by the machine z.

Proof. The idea is that we can firstly use sorting (See Theorem 2.3.1) to make queries and the corresponding values be stored consecutively in several machines. The issue remaining is that there may be many queries queried the same position such that some queries may not be stored in the machine which holds the corresponding value. In this case, we need to find the predecessor by invoking the algorithm shown in Theorem 2.3.3. □

The Multiple queries algorithm is shown as the following:

Multiple Queries Algorithm:

- **Setups:**
  - There are $3p = \Theta(n^\delta)$ machines indexed from 1 to $3p$ each with local memory size $s = \Theta(n^\delta)$.
  - The machine with index from $2p + 1$ to $3p$ are input/output machines.
  - Sets $S_1, S_2, \cdots, S_k, Q$ are stored on machine $2p + 1$ to $3p$. ▹ Corresponding to Lemma 2.3.6
- **The first round:**
  - Machine $i \in \{2p + 1, \cdots, 3p\}$ scans its local memory, and send all the tuples with form ("f_j", (x, y)) or ("Q", (x, y, z)) to machine $i - p$, where "f_j" is the name of f_j (also S_j) for $j \in [t]$. Until the end of the computation, machine i sends itself messages to keep its local data.
- **Using constant number ($O(1/\delta)$) of rounds to sort:**
  - Use machine 1 to $2p$ to sort all the tuples stored on machine $p + 1$ to $2p$, and thus at the end of this stage, machine $p + 1$ to $2p$ holds sorted tuples. For tuple with the form ("f_j", (x, y)), the first key value is "f_j", the second key value is x and the third key value is $-\infty$ which is the highest priority. For tuple with form ("Q", (x, y, z)), the first key value is x, the second key value

is y, and the third key value is ∞ which is the lowest priority. The comparison in the sorting is: Firstly compare the first key. If they are the same, then compare the second key. If they are still the same, compare the third key.

- **Using constant number ($O(1/\delta)$) of rounds to find predecessors:**
  - Machine $p+1$ to $2p$ scans its local memory. For a tuple in the form ("f_j", (x, y)), the machine marked it as "1". For a tuple in the form ("Q", (x, y, z)), the machine marked it as "0".
  - Machine 1 to $2p$ together invoke the Predecessor algorithm (Theorem 2.3.3), where the input is on machine $p+1$ to machine $2p$.
- **The last round:**
  - Machine $p+1$ to $2p$ scans its local memory. For each tuple with form ("Q", (x, y, z)), it sends machine z a tuple ("Q'", (x, y, z, w)), where x is the name of f_j, and $w = f_j(y)$.

2.3.5 Sequence operations

In this section, we introduce some MPC model operations for sequence.

Sequence standardizing. Suppose there is a sequence A, and one of its set representation (see Definition 2.3.5) S is stored in the system. The goal is to modify the set S such that S is a standard set representation of A.

We can compute the index (see **Indexing elements in sets** in Section 2.3.3) of elements in S. Then for each element $(x, y) \in S$, we can query (see **Multiple queries** in Section 2.3.4) the index of (x, y) in S. Suppose the index is i, we modify the tuple ("S", (x, y)) to ("S", (i, y)).

Sequence duplicating. Suppose there is a sequence $A = (a_1, a_2, \cdots, a_s)$, and one of its set representation (see Definition 2.3.5) S is stored in the system. Furthermore, there is a mapping $f : [s] \to \mathbb{Z}_{\geq 0}$ which is also stored in the system. The goal is to get a set S' stored in the system

such that S' is a set representation of the sequence:

$$(\underbrace{a_1, a_1, \cdots, a_1}_{f(1) \text{ times}}, \underbrace{a_2, a_2, \cdots, a_2}_{f(2) \text{ times}}, \cdots, \underbrace{a_s, a_s, \cdots, a_s}_{f(s) \text{ times}}).$$

Firstly, we can standardize (see the above paragraph **Sequence standardizing**) the set S. Then for each tuple $(\text{“}S\text{”}, (i, a_i))$, we create a tuple $(\text{“}S_i\text{”}, a_i)$, and we can query (see **Multiple queries** in Section 2.3.4) the value of $f(i)$. Then we can copy (see **Copies of sets** in Section 2.3.3) set S_i $f(i)$ times. For each tuple $(\text{“}S_{i,j}\text{”}, a_i)$, we create a tuple $(\text{“}S'\text{”}, ((i, j), a_i))$. Then we can compute the index (see **Indexing elements in sets** in Section 2.3.3) of each element in S'. For each tuple $(\text{“}S'\text{”}, ((i, j), a_i))$, we can query (see **Multiple queries** in Section 2.3.4) the index i' of it, and then modify the tuple as $(\text{“}S'\text{”}, (i', a_i))$.

Sequence insertion. Suppose there are $k + 1$ sequences $A = (a_1, a_2, \cdots, a_s), A_1, \cdots, A_k$ which have sets representations (see Definition 2.3.5) $S, S_1, \cdots, S_k$ respectively and stored on the system. There is also a mapping $f : [k] \to \{0\} \cup [s]$ stored on the system where $\forall i \neq j \in [k], f(i) \neq f(j)$. The goal is to insert each sequence A_i into the sequence A, and A_i should be between the element $a_{f(i)}$ and $a_{f(i)+1}$.

Firstly, we can standardize (see **Sequence standardizing** in Section 2.3.3) S. Then we can compute an upper bound of the total size (see **Sizes of sets** in Section 2.3.3) $N = |S| + |S_1| + \cdots + |S_k| + 1$. For each tuple $(\text{“}S\text{”}, (i, a_i))$, we can modify it as $(\text{“}S\text{”}, (i \cdot N, a_i))$. For each tuple $(\text{“}S_i\text{”}, (j, a_{ij}))$, we query (see **Multiple queries** in Section 2.3.4) the value of $f(i)$, then create a tuple $(\text{“}S\text{”}, (f(i) \cdot N + j, a_{ij}))$.

2.3.6 Multiple tasks

In this section, we show that if the entire computational task consists of some independent small computational tasks, we are able to schedule the machines such that the small computational tasks can be computed simultaneously.

Task and multiple tasks problem. A computational task here is running a specific algorithm on specific input data.

There are k sets $S_1, S_2, \cdots, S_k$ stored in the system. Let $n = \sum_{i=1}^{k} |S_i|$ be the total input size. There are h independent computational tasks $T_1, T_2, \cdots, T_h$. Each task T_i needs to take some sets $\mathcal{S}_i \subseteq \{S_1, S_2, \cdots, S_k\}$ as its input, and is running a $(\gamma_i, \delta_i) -$ MPC algorithm in r_i rounds where $\gamma_i \in \mathbb{R}_{\geq 0}$, constant $\delta_i \in (0, 1)$. $\forall i \in [h]$, let $n_i = \sum_{S \in \mathcal{S}_i} |S|$ be the input size of task T_i. Without loss of generality, we can assume that the input of different tasks are disjoint. Otherwise we can use sets copying technique (See Section 2.3.3) to generate different copies of input sets for the tasks shared the same input set. The goal here is to use the small number of rounds to finish all the tasks. Since we can always use sorting and indexing to extract the desired input data. The most naive way is to compute the tasks one-by-one. This can be trivially done in $r = O(\sum_{i=1}^{h} r_i)$ rounds in $(\gamma, \delta) -$ MPC model for $\gamma = \log_n(h) + \max_{i \in [h]} \gamma_i, \delta = \max_{i \in [h]} \delta_i$. Here we show how to compute all the tasks simultaneously in $r = O(\max_{i \in [h]} r_i)$ rounds in $(\gamma, \delta) -$ MPC model for $\gamma = \log_n(m) - 1, \delta = \max_{i \in [h]} \delta_i$, where $m = \Theta(n + \sum_{i=1}^{h} n_i^{1+\gamma_i})$.

Each machine scans its local memory. If the machine holds a tuple $(\text{“}S_i\text{”}, x)$, and S_i is a part of input of task T_j, then it creates a tuple $(\text{“}W_j\text{”}, (\text{“}S_i\text{”}, x))$. Thus, at the end of this step, there are additional h sets $W_1, W_2, \cdots, W_h$ stored in the system. Here $W_i, i \in [h]$ contains all the information of input data of task T_i. Then we can compute a mapping f such that $\forall i \in [h], f(W_i) = |W_i|$ (see Section 2.3.3). Thus, we know the input size of each task. Then each machine scans its local memory. If the machine holds a tuple $(\text{“}f\text{”}, (\text{“}W_i\text{”}, |W_i|))$, then it creates a tuple $(\text{“}H_i\text{”}, |W_i|)$, i.e. a set $H_i = \{|W_i|\}$. Then for each set $H_i = \{|W_i|\}, i \in [h]$, we can copy (see Section 2.3.3) it $s_i = c \cdot |W_i|^{1+\gamma_i}$ times for a sufficiently large c to get sets $H_{i,1} = H_{i,2} = \cdots = H_{i,s_i} = \{|W_i|\}$. Each set $H_{i,j}$ is just a placeholder of one unit working space of the task T_i. Thus, the number of copies of the set H_i is the total space needed for the task T_i. We can sort all the tuples $(\text{“}H_{i,j}\text{”}, |W_i|)$ on machines with index in $I = \{2, 5, 8, 11, \cdots, 3p - 1\}$, where local memory $s = \Theta(n^{\delta})$, total required memory $m = \Theta(n + \sum_{i=1}^{h} n_i^{1+\gamma_i})$, and $p = \Theta(m/s)$ For each machine with index $q \in I$, the tuples on

that machine must be in the following form

$$(\text{“}H_{i,j}\text{”}, |W_i|), (\text{“}H_{i,j+1}\text{”}, |W_i|), \cdots, (\text{“}H_{i,s_i}\text{”}, |W_i|), (\text{“}H_{i+1,1}\text{”}, |W_{i+1}|), \cdots, (\text{“}H_{i+1,s_{i+1}}\text{”}, |W_{i+1}|),$$
$$(\text{“}H_{i+2,1}\text{”}, |W_{i+2}|), \cdots, (\text{“}H_{i+2,s_{i+2}}\text{”}, |W_{i+2}|), \cdots, (\text{“}H_{i',1}\text{”}, |W_{i'}|), \cdots (\text{“}H_{i',j'}\text{”}, |W_{i'}|).$$

Then machine q just sends all the tuples $(\text{“}H_{i,j}\text{”}, |W_i|), (\text{“}H_{i,j+1}\text{”}, |W_i|), \cdots, (\text{“}H_{i,s_i}\text{”}, |W_i|)$ to machine $q-1$, and sends all the tuples $(\text{“}H_{i',1}\text{”}, |W_{i'}|), (\text{“}H_{i',2}\text{”}, |W_{i'}|), \cdots (\text{“}H_{i',j'}\text{”}, |W_{i'}|)$ to machine $q+1$. Thus, $\forall i \in [h]$,

1. either all the $H_{i,1}, H_{i,2}, \cdots, H_{i,s_i}$ are stored on consecutive machines, machine q to machine q', and any of machine q to machine q' does not hold other tuples,

2. or there is a unique machine q which holds all the sets $H_{i,1}, H_{i,2}, \cdots, H_{i,s_i}$.

For each machine $q \in [3p]$, if $H_{i,1}$ is held by machine q, then it creates a tuple $(\text{“st”}, (\text{“}T_i\text{”}, q))$. If H_{i,s_i} is held by machine q, then it creates a tuple $(\text{“ed”}, (\text{“}T_i\text{”}, q))$. The mapping st, ed then are stored in the system, where $\text{st}(T_i)$ is the index of the first machine assigned to task T_i, and $\text{ed}(T_i)$ is the index of the last machine assigned to task T_i. Recall that W_i contains all the information of the input data to task T_i. The remaining task is to move the input data of task T_i to the machines with index from $\text{st}(T_i)$ to $\text{ed}(T_i)$. According to Section 2.3.3, we can compute a mapping f', such that $f'(W_i, x)$ records the index of $x \in W_i$ in set W_i. Now, each machine scans its local memory. For each tuple $(\text{“}W_j\text{”}, (\text{“}S_i\text{”}, x))$, the machine needs to query the value of $f'(W_j, (\text{“}S_i\text{”}, x))$, the value of $\text{st}(T_j)$ and the value of $\text{ed}(T_j)$. By Lemma 2.3.6, these queries can be handled simultaneously in constant number of rounds. Then the machine can send the tuple $(\text{“}S_i\text{”}, x)$ to the corresponding machine based on the value of $f'(W_j, (\text{“}S_i\text{”}, x))$, $\text{st}(T_j)$, and $\text{ed}(T_j)$. Finally, $\forall i \in [h]$, since $\delta \geq \delta_i$ and $(\text{ed}(T_i) - \text{st}(T_i) + 1) \cdot s = \Theta(n_i^{1+\gamma_i})$, the machines with index from $\text{st}(T_i)$ to $\text{ed}(T_i)$ can simulate task T_i in r_i number of rounds.

Chapter 3: Some General Techniques

In this chapter, we will see some general techniques for designing other efficient graph algorithms. In Section 3.1, the *truncated broadcasting* technique is introduced. In Section 3.2, we describe the approach of *double-exponential speed problem size reduction*.

3.1 Truncated broadcasting

In this section, we introduce a procedure which uses a small number of iterations to explore b-closest vertices of every vertex. This procedure was originally used in [31]. Here we re-discover the procedure and show the relationship between the number of iterations and the hop diameter of the graph. The detailed procedure is described in Algorithm 1.

Algorithm 1 Truncated Broadcasting

procedure TRUNCATEDBROADCASTING($G = (V, E, w), b$)
  For $v \in V$, initialize a list $L^{(0)}(v)$ containing all b-closest neighbors (including v itself) of v.
  For $v \in V, u \in L^{(0)}(v)$, let $\mathrm{dist}^{(1)}(v,u) \leftarrow w(v,u)$.
  Let $i \leftarrow 0$.
  for true do
    $i \leftarrow i + 1$.
    For $v, u \in V$, initialize $\mathrm{dist}^{(2^i)}(v,u) \leftarrow \infty$.
    For each vertex $v \in V$, each vertex $x \in L^{(i-1)}(v)$ and each vertex $u \in L^{(i-1)}(x)$, if $\mathrm{dist}^{(2^{i-1})}(v,x) + \mathrm{dist}^{(2^{i-1})}(x,u) < \mathrm{dist}^{(2^i)}(v,u)$, update $\mathrm{dist}^{(2^i)}(v,u) \leftarrow \mathrm{dist}^{(2^{i-1})}(v,x) + \mathrm{dist}^{(2^{i-1})}(x,u)$.
    For $v \in V$, add u to list $L^{(i)}(v)$ if $\mathrm{dist}^{(2^i)}(v,u)$ is one of the b smallest values among $\mathrm{dist}^{(2^i)}(v,x)$ for $x \in V$. If there is a tie, take the vertex with a smaller label.
    If $\forall v \in V, L^{(i)}(v) = L^{(i-1)}(v)$ and $\forall u \in L^{(i)}(v), \mathrm{dist}^{(2^i)}(u,v) = \mathrm{dist}^{(2^{i-1})}(v,u)$, break the loop.
  end for
  $t \leftarrow i$.
  Output $S_v \leftarrow L^{(t)}(v)$ for each $v \in V$ and output $\mathrm{dist}(v,u) \leftarrow \mathrm{dist}^{(2^t)}(v,u)$ for each $v \in V, u \in S_v$.
end procedure

Lemma 3.1.1. *Let $G = (V, E, w)$ be an undirected weighted graph with weights $w : E \to \mathbb{R}_{>0}$. Let $b \in \mathbb{Z}_{\geq 1}$. Consider the procedure* TRUNCATEDBROADCASTING(G, b). *(Algorithm 1).*

1. *The output S_v for each vertex $v \in V$ contains all b-closest vertices of v and the size $|S_v| \leq b$.*

2. *The output* $\mathrm{dist}(v,u)$ *for each vertex $v \in V$ and each $u \in S_v$ satisfies* $\mathrm{dist}(v,u) = \mathrm{dist}_G(v,u)$.

3. *The number of iterations t is at most* $\min(\lceil \log(\mathrm{diam}(G)) \rceil, \lceil \log(b) \rceil) + 1$.

Proof. The proof is by induction. We claim that $L^{(i)}(v)$ is exactly the set of all b-closest vertices of v under 2^i-hop distance. Furthermore, $\forall u \in L^{(i)}(v), \mathrm{dist}^{(2^i)}(v,u) = \mathrm{dist}_G^{(2^i)}(v,u)$.

Claim 3.1.2. $\forall i \in \{0,1,\cdots,t\}, \forall v \in V, L^{(i)}(v) = \{u \in V \mid |\{x \in V \mid \mathrm{dist}_G^{(2^i)}(v,x) < \mathrm{dist}_G^{(2^i)}(v,u)$ *or* $\mathrm{dist}_G^{(2^i)}(v,x) = \mathrm{dist}_G^{(2^i)}(v,u)$ *and x has a smaller label than u}$| < b\}$, *and* $\forall u \in L^{(i)}(v), \mathrm{dist}^{(2^i)}(v,u) = \mathrm{dist}_G^{(2^i)}(v,u)$.

Proof. The proof is by induction. When $i = 0$, by the construction of $L^{(0)}(v)$, $L^{(0)}(v)$ contains all b-closest direct neighbors of v and $\forall u \in L^{(0)}(v), \mathrm{dist}^{(2^i)}(v,u) = w(v,u) = \mathrm{dist}_G^{(1)}(v,u)$. Thus, the claim holds for $i = 0$. Suppose the claim holds for $i-1$. Consider a vertex u which is a b-closest vertex of v under 2^i-hop distance in G. Consider a 2^i-hop shortest path from v to u, by optimality, there must exists a vertex x such that x is a b-closest vertex of v under 2^{i-1}-hop distance in G and u is a b-closest vertex of x under 2^{i-1}-hop distance in G. By induction hypothesis, $x \in L^{(i-1)}(v)$, $u \in L^{(i-1)}(x)$, $\mathrm{dist}^{(2^{i-1})}(v,x) = \mathrm{dist}_G^{(2^{i-1})}(v,x)$ and $\mathrm{dist}^{(2^{i-1})}(x,u) = \mathrm{dist}_G^{(2^{i-1})}(x,u)$. According to line 8-9 of Algorithm 1, we can verify that $L^{(i)}(v)$ is exactly the set of all b-closest vertices of v under 2^i-hop distance and $\forall u \in L^{(i)}(v), \mathrm{dist}^{(2^i)}(v,u) = \mathrm{dist}_G^{(2^i)}(u,v)$. □

Let $h = \min(\mathrm{diam}(G), b)$. For each vertex $v \in V$ and for each b-closest vertex u of v, the h-hop distance between u and v is exact the distance between u and v. Thus, according to Claim 3.1.2, if $i > \lceil \log h \rceil$, $\forall v \in V$, we have $L^{(i)}(v) = L^{(i-1)}(v)$ and $\forall u \in L^{(i)}(v), \mathrm{dist}^{(2^i)}(v,u) = \mathrm{dist}^{(2^{i-1})}(v,u)$. Therefore, the number of iterations t is at most $\min(\lceil \log(\mathrm{diam}(G)) \rceil, \lceil \log(b) \rceil) + 1$.

Imagine that we do not break the loop at line 10 of Algorithm 1 and continue to generate $L^{(t+1)}(v), L^{(t+2)}(v), \cdots, L^{(t')}(v)$ for every $v \in V$ where $t' = \lceil \log(h) \rceil + 2$. According to the proof of Claim 3.1.2, for $v \in V, L^{(t')}(v)$ is exactly the set of all b-closest vertices of v and for $v \in V, u \in L^{(t')}(v), \mathrm{dist}^{(2^{t'})}(v,u) = \mathrm{dist}_G^{(2^t)}(v,u) = \mathrm{dist}_G(v,u)$. Since $\forall v \in V, L^{(t)}(v) = L^{(t-1)}(v)$ and

$\forall u \in L^{(t)}(v), \text{dist}^{(2^t)}(u,v) = \text{dist}^{(2^{t-1})}(v,u)$, we have $\forall v \in V, L^{(t)}(v) = L^{(t+1)}(v) = \cdots = L^{(t')}(v)$ and $\forall u \in L^{(t')}(v), \text{dist}^{(2^t)}(u,v) = \text{dist}^{(2^{t+1})}(u,v) = \cdots = \text{dist}^{(2^{t'})}(v,u)$. Thus, for $v \in V, S_v = L^{(t)}(v)$ is exactly the set of all b-closest vertices of v and for $v \in V, u \in S_v, \text{dist}(v,u) = \text{dist}^{(2^t)}(v,u) = \text{dist}_G^{(2^t)}(v,u) = \text{dist}_G(v,u)$. □

3.1.1 Implementation in parallel computing models

Now let us consider how to implement Algorithm 1 in parallel setting. Suppose that the input graph G contains n vertices and m edges, and the parameter $b \in [n]$.

PRAM. In the PRAM model, if we do not optimize the depth and work, Algorithm 1 can be simply implemented in $\text{poly}(\log n)$ depth and $\widetilde{O}(m+nb^2)$ total work. This is already shown by [31].

Lemma 3.1.3 ([31])**.** *Given an n-vertex m-edge undirected weighted graph $G = (V, E, w)$ and a parameter $b \in [n]$,* TRUNCATEDBROADCASTING(G, b) *(Algorithm 1) can be implemented in PRAM with $\log^{O(1)} n$ depth and $\widetilde{O}(nb^2 + m)$ work.*

The PRAM implementation is described as the following.

1. For $v \in V$, initialize a list $L^{(0)}(v)$ containing b closest neighbors (including v itself) of v. For $u \in L^{(0)}(v)$, compute $\text{dist}^{(1)}(v,u) \leftarrow w(v,u)$. This step can be done using PRAM sorting in $\text{poly}(\log n)$ depth and $\widetilde{O}(m+n)$ work. Let $t \leftarrow \lceil \log n \rceil$.

2. For $i = 1 \rightarrow t$:

   (a) For $v, u \in V$, (conceptually) initialize $\text{dist}^{(2^i)}(v,u) \leftarrow \infty$.

   (b) Assign b^2 processors for each $v \in V$. Each processor reads a vertex $x \in L^{(i-1)}(v)$ and then reads a vertex $u \in L^{(i-1)}(x)$. If $\text{dist}^{(2^{i-1})}(v,x) + \text{dist}^{(2^{i-1})}(x,u) < \text{dist}^{(2^i)}(v,u)$, update $\text{dist}^{(2^i)}(v,u) \leftarrow \text{dist}^{(2^{i-1})}(v,x) + \text{dist}^{(2^{i-1})}(x,u)$.

   (c) For $v \in V$, add u to list $L^{(i)}(v)$ if $\text{dist}^{(2^i)}(v,u)$ is one of the b smallest values among $\text{dist}^{(2^i)}(v,x)$ for $x \in V$. If there is a tie, take the vertex with a smaller label.

3. Output $L^{(t)}(v)$ for each vertex $v \in V$ and output $\text{dist}^{(2^t)}(v,u)$ for each $u \in L^{(t)}(v)$.

MPC. In the MPC model, we want the number of rounds to be $O(\min(\lceil\log(\mathrm{diam}(G))\rceil, \lceil\log(b)\rceil))$. Thus, the detailed implementation is more involved than the PRAM implementation.

Lemma 3.1.4. *Let graph $G = (V, E, w), n = |V|, N = |V| + |E|, b \in [n]$ and $m = \Theta(N^\gamma)$ for some arbitrary $\gamma \in [0, 2]$. If $b^2 \leq m$,* TruncatedBroadcasting(G, b) *(Algorithm 1) can be implemented in $(\gamma, \delta) -$* MPC *model for any constant $\delta \in (0, 1)$. Furthermore, the parallel running time is $O(t)$, where $t \leq \min(\lceil\log(\mathrm{diam}(G))\rceil, \lceil\log(b)\rceil) + 1$ is the number of iterations (see Lemma 3.1.1) of* TruncatedBroadcasting(G, b).

Proof. According to Section 2.3. The graph is stored in the system as the edge set E and the weight mapping w, i.e., $\forall\{u, v\} \in E$, there is a unique tuple $(\text{“}E\text{”}, \{u, v\})$ and a unique tuple $(\text{“}w\text{”}, (\{u, v\}, w(u, v)))$ stored in the machines. In the remaining of the proof, we refer readers to Section 2.3 for all basic MPC operations. In the remaining of the proof, we will discuss how to implement Algorithm 1 in the MPC model.

To implement line 2, we can create a tuple $(\text{“}L^{(0)}(v)\text{”}, u)$ for each tuple $(\text{“}E\text{”}, \{v, u\})$. We also add $(\text{“}L^{(0)}(v)\text{”}, v)$ for each $v \in V$. Furthermore, we can query $w(u, v)$ for each tuple $(\text{“}L^{(0)}(v)\text{”}, u)$ (see **Multiple queries**). Then we can sort tuples $(\text{“}L^{(0)}(v)\text{”}, u)$ via $w(u, v)$: we first compare the key “$L^{(0)}(v)$”, then compare the key $w(u, v)$, and finally compare the key label u (see Theorem 2.3.1). Then for each tuple $(\text{“}L^{(0)}(v)\text{”}, u)$ we can compute the index (see **Indexing elements in sets** and **Multiple queries**) of u in set $L^{(0)}(v)$. If the index of u in set $S_v^{(0)}$ is larger than b, then delete u from $L^{(0)}(v)$, i.e. delete the tuple $(\text{“}L^{(0)}(v)\text{”}, u)$.

To implement line 3, for each tuple $(\text{“}L^{(0)}(v)\text{”}, u)$ survived, we query $w(v, u)$ (see **Multiple queries**) and create a tuple $(\text{“ dist}^{(1)}\text{ ”}, ((v, u), w(v, u)))$.

Next, let us discuss how to implement line 8. We can compute the size of every set stored in the system (see **Sizes of sets**). Then for each tuple $(\text{“}L^{(i-1)}(v)\text{”}, u)$, the corresponding machine queries (see **Multiple queries**) the size of $L^{(i-1)}(u)$. For each tuple $(\text{“}L^{(i-1)}(v)\text{”}, x)$ we create a tuple $(\text{“target}_x^i\text{”}, v)$. Thus, $v \in \text{target}_x^i$ means that we try to update the distance from v to vertices in $L^{(i-1)}(x)$. $|\text{target}_x^i|$ means that $L^{(i-1)}(x)$ needs to copy $|\text{target}_x^i|$ times. For each tuple $(\text{“}L^{(i-1)}(x)\text{”}, u)$, we query (see **Multiple queries**) the size (see **Sizes of sets**) of target_x^i. Then we can copy (see

Copies of sets) each set $L^{(i-1)}(x)$ $|\text{target}^i_x|$ times. Then for each tuple ("target^i_x", v), we can query (see **Multiple queries**) the index p (see **Indexing elements in sets**) of v in set target^i_x, and then create a tuple ("f^i", (("target^i_x", p), v)) which means that the p^{th} element of target^i_x is $f^i(\text{target}^i_x, p) = v$. For each tuple ("$L^{(i-1)}(x)_j$", u), we query (see **Multiple queries**) the value $v = f^i(\text{target}^i_x, j)$, and then create a tuple ("$L^{(i)}(v)$", u). Furthermore, we also query (see **Multiple queries**) $\text{dist}^{(2^{i-1})}(v, x)$ and $\text{dist}^{(2^{i-1})}(x, u)$, and create a tuple ("$\text{dist}^{(2^i)}$", $((v, u), \text{dist}^{(2^{i-1})}(v, x) + \text{dist}^{(2^{i-1})}(x, u))$). We then remove the duplicates (see **Duplicates removing**) of elements of for every set $L^{(i)}(v)$. We can use sorting (see Theorem 2.3.1) to sort tuples ("$\text{dist}^{(2^i)}$", $((v, u), \text{dist}^{(2^{i-1})}(v, x) + \text{dist}^{(2^{i-1})}(x, u))$) and for each (v, u), we only keep one tuple ("$\text{dist}^{(2^i)}$", $((v, u), \text{dist}^{(2^{i-1})}(v, x) + \text{dist}^{(2^{i-1})}(x, u))$) for x with the minimum $\text{dist}^{(2^{i-1})}(v, x) + \text{dist}^{(2^{i-1})}(x, u)$ and remove other tuples.

Let us consider how to implement line 9. For each tuple ("$L^{(i)}(v)$", u), we query (see **Multiple queries**) $\text{dist}^{(2^i)}(v, u)$ and sort (see Theorem 2.3.1) tuples ("$L^{(i)}(v)$", u) via the queried value $\text{dist}^{(2^i)}(v, u)$. For the sorting, the tuple ("$L^{(i)}(v)$", u) has smaller rank if it has smaller $\text{dist}^{(2^i)}(v, u)$. When there is a tie, a tuple ("$L^{(i)}(v)$", u) with smaller label u has smaller rank. Then, we can query (see **Multiple queries**) the index (see Indexing elements in sets) of u in $L^{(i)}(v)$. If the index is larger than b, we remove u from $L^{(i)}(v)$, i.e., delete the tuple ("$L^{(i)}(v)$", u).

Finally, we discuss how to implement line 10. For each tuple ("$L^{(i)}(v)$", u), query (see **Multiple queries**) whether $u \in L^{(i-1)}(v)$ (see **Set membership**). For each tuple ("$L^{(i-1)}(v)$", u), query (see **Multiple queries**) whether $u \in L^{(i)}(v)$ (see **Set membership**). If $\exists v \in V$ such that $\exists u \in L^{(i-1)}(v) \setminus L^{(i)}(v)$ or $\exists u \in L^{(i)}(v) \setminus L^{(i-1)}(v)$, create a tuple ("*Undone*", v). Every machine queries (see **Multiple queries**) the size (see **Sizes of sets**) of Undone. If it is not empty, then all machines know that they should continue the loop. Otherwise, for each tuple ("$L^{(i)}(v)$", u), query (see **Multiple queries**) $\text{dist}^{(2^i)}(v, u)$ and $\text{dist}^{(2^{i-1})}(v, u)$. If $\text{dist}^{(2^i)}(v, u) \neq \text{dist}^{(2^{i-1})}(v, u)$, create a tuple ("*Undone*", v). Every machine queries (see **Multiple queries**) the size (see **Sizes of sets**) of Undone. If it is empty, then all machines know that they should break the loop.

In the i^{th} iteration, we only need to maintain sets $V, L^{(i-1)}(v), L^{(i)}(v)$ and mappings $\text{dist}^{(2^{i-1})}$ and $\text{dist}^{(2^i)}$. Since all the copy operation will create at most $n \cdot b^2 \leq m$ tuples, the total space

needed is $\Theta(m)$ plus the space needed to maintain sets $V, E, L^{(i-1)}(v), L^{(i)}(v)$ and mappings $\text{dist}^{(2^{i-1})}$ and $\text{dist}^{(2^i)}$. According to Lemma 3.1.1, the total space needed to store all $L^{(i-1)}(v), L^{(i)}(v)$ and $\text{dist}^{(2^{i-1})}, \text{dist}^{(2^i)}$ is at most $O(n \cdot b)$. Thus, the total space is at most $\Theta(m)$.

It is easy to verify that the above implementation shows that the parallel time is $O(t)$, where $t \leq \min(\lceil \log(\text{diam}(G)) \rceil, \lceil \log(b) \rceil) + 1$ is the number of iterations (see Lemma 3.1.1) of Algorithm 1. □

3.2 Double-exponential speed problem size reduction

Double-exponential speed problem size reduction is a general technique which solves a problem in $O(\log \log n)$ iterations using small computational resource. To be more precise, for any problem characterized by a size parameter n, if there is a subroutine which uses total computational resource (e.g., space or running time) $\Theta(m)$ to reduce the problem size such that the reduced problem size is n/k for $k = (m/n)^{\Omega(1)}$, then we can solve the problem in $O(m)$ total computational resource by iteratively calling the subroutine $O(\log \log n)$ times. The proof is sketched as the following. When the problem size is $O(1)$, we can solve the problem easily. Let n_i be the problem size after the i-th iteration of calling the subroutine. Suppose the subroutine uses $\Theta(m)$ computational resource to reduce the problem size to at most n_i/k_i for $k_i = (m/n_i)^c, c = \Omega(1)$. Then, after repeating calling the subroutine i times, the problem size is $n_i \leq n_{i-1}/(m/n_{i-1})^c \leq n \cdot (n/m)^{(1+c)^i - 1}$. Thus, after $O(\log_{1+c} \log_{m/n} n)$ iterations, the problem size will be reduced to $O(1)$.

We will see concrete applications of this technique in graph connectivity, spanning forest, shortest path algorithms in later chapters.

Chapter 4: Graph Connectivity and Spanning Forest

In this chapter, we will show how to use truncated broadcasting and double-exponential speed problem size reduction techniques introduced in Chapter 3 to design graph connectivity and spanning forest algorithms. Our algorithms can be implemented in parallel models with small number of rounds and small total space/work.

4.1 Overview of techniques

Graph Connectivity: A natural approach to the graph connectivity problem is via the classic primitive of contracting to leaders: select a number of leader verteces, and contract every vertex (or most vertices) to a leader from its connected component (this is usually implemented by labeling the vertex by the corresponding leader). Indeed, many previous algorithms (see e.g. [1, 76, 47]) are based on this approach. There are two general questions to address in this approach: 1) how to choose leader vertices, and 2) how to label each vertex by its leader. For example, the algorithm in [1] randomly chooses half of the vertices as leaders, and then contracts each non-leader vertex to one of its neighbor leader vertex. Thus, in each round of their algorithm, the number of vertices drops by a constant fraction. At the same time, half of the vertices are leaders, and hence their algorithm still needs at least $\Omega(\log n)$ rounds to contract all the vertices to one leader. Note that a constant fraction of leaders is needed to ensure that there is a constant fraction of non-leader vertices who are adjacent to at least one leader vertex and hence are contracted. This leader selection method appears optimal for some graphs, e.g. path graphs.

To improve the runtime to $\ll \log n$, one would have to choose a much smaller fraction of the vertices to be leaders. Indeed, for a graph where every vertex has a large degree, say at least $d \gg \log n$, we can choose fewer leaders: namely, we can choose each vertex to be a leader with

probability $p = \Theta((\log n)/d)$. Then the number of leaders will be about $\widetilde{O}(n/d)$, while each non-leader vertex has at least one leader neighbor with high probability. After contracting non-leader vertices to leader vertices, the number of remaining vertices is only a $1/d$ fraction of original number of vertices.

By the above discussion, the goal would now be to modify our input graph G so that every vertex has a uniformly large degree, without affecting the connectivity of the graph. An obvious such modification is to add edges between pairs of vertices that are already in the same connected component. In particular, if a vertex v learns of a large number of vertices which are in the same connected component as v, then we can add edges between v and those vertices to increase the degree of v. A naïve way to implement the latter is via broadcasting: each vertex v first initializes a set S_v which contains all the neighbors of v, and then, in each *round*, every vertex v updates the set S_v by adding the union of the sets S_u over all neighbors u of v (old and new). This approach takes log-diameter number of rounds, and each vertex learns all vertices which are in the same connected component at the end of the procedure. However, in a single round, the total communication needed may be as huge as $\Omega(n^3)$ since each of n vertices may have $\Omega(n)$ neighbors, each with a set of size $\Omega(n)$.

Since our goal of each vertex v is to learn only d vertices in the same component (not necessarily the entire component), we can therefore use a "truncated" version of the above broadcasting procedure. We can use the truncated broadcasting (Algorithm 1) technique introduced in Chapter 3 to achieve this goal. At the end of truncated broadcasting procedure, each vertex v learns a set S_v which either contains at least b vertices which are in the same connected component as v or contains all vertices in the same connected component as v. The truncated broadcasting procedure takes at most log-diameter rounds. Furthermore, the total communication needed is at most $O(n \cdot d^2)$.

Our full graph connectivity algorithm calls the truncated broadcasting procedure iteratively, for values d that follow a certain "schedule", depending on the available space. At the beginning of the algorithm, we have an n vertex graph G with diameter D, and a total of $\Omega(m)$ space. The

algorithm proceeds in phases, where each phase takes $O(\log D)$ rounds of communication. In the first phase, the starting number of vertices is $n_1 = n$. We implement a truncated broadcasting procedure where the target degree d is $d_1 = (m/n_1)^{1/2}$, using $O(\log D)$ rounds and $O(m)$ total space. Then we can randomly select $\widetilde{O}(n_1/d_1)$ leaders, and contract all the non-leader vertices to leader vertices. At the end of the first phase, the total number of remaining vertices is at most $n_2 = \widetilde{O}(n_1/d_1) = \widetilde{O}(n_1^{1.5}/m^{0.5})$. In general, suppose, at the beginning of the i^{th} phase, the number of remaining vertices is n_i. Then we use the truncated broadcasting procedure for value d set to $d_i = (m/n_i)^{1/2}$, thus making each vertex have degree at least $d_i = (m/n_i)^{1/2}$ in $O(\log D)$ number of communication rounds and $O(m)$ total space. Then we choose $\widetilde{O}(n_i/d_i)$ leaders, and, after contracting non-leaders, the number n_{i+1} of remaining vertices is at most $\widetilde{O}(n_i^{1.5}/m^{0.5})$. Let us look at the progress of the value d_i. We have that $d_{i+1} = \widetilde{\Omega}((m/n_{i+1})^{1/2}) = \widetilde{\Omega}((m^{1.5}/n_i^{1.5})^{1/2}) = \widetilde{\Omega}(d_i^{1.5})$. Thus, we are making double exponential progress on d_i, which implies that the total number of phases needed is at most $O(\log \log_{m/n} n)$, and the total parallel time is thus $O(\log D \cdot \log \log_{m/n} n)$. This is an example of application of double-exponential speed problem size reduction technique introduced by Chapter 3.

Spanning Forest: Extending a connectivity algorithm to a spanning forest algorithm is usually straightforward. For example, in [1], they only contract a non-leader vertex to an adjacent leader vertex, thus their algorithm can also give a spanning forest, using the contracted edges. Here however, extending our connectivity algorithm to a spanning forest algorithm requires several new ideas. In our connectivity algorithm, because of the added edges, we only ensure that when a vertex u is contracted to a vertex v, u and v must be in the same connected component; but u and v may not be adjacent in the original graph. Thus, we need to record more information to help us build a spanning forest.

We can represent a forest as a collection of parent pointers $\text{par}(v)$, one for each vertex $v \in V$. If v is a root in the forest, then we let $\text{par}(v) = v$. We use $\text{dep}_{\text{par}}(v)$ to denote the depth of v in the forest, i.e. $\text{dep}_{\text{par}}(v)$ is the distance from v to its root. Let $\text{dist}_G(u, v)$ denote the distance between

two vertices u and v in a graph G.

Recall that our connectivity algorithm uses the truncated broadcasting procedure. According to the truncated broadcasting procedure, we know the distance from v to each vertex in S_v. Thus we are able to compute a local shortest path tree for S_v with root v: for each vertex $u \in S_v$, find an arbitrary vertex $x \in S_v$ such that $x \in S_v, \{u, x\} \in E$ and $\text{dist}_G(v, u) = \text{dist}_G(v, x) + 1$, and let x be the parent of u in the local shortest path tree. Thus, we are able to find n local shortest path trees where there is a tree with root v for each vertex v. Next, we show how to use these n local shortest path trees to construct a forest with the roots in the forest being the leaders.

As discussed in the connectivity algorithm, if every local shortest path tree has size at least d, we can choose each vertex as a leader with probability $p = \Theta((\log n)/d)$ and then every tree will contain at least one leader with high probability. Let L be the set of sampled leaders, and let $\text{dist}_G(v, L)$ be defined as $\min_{u \in L} \text{dist}_G(v, u)$. Consider a non-leader vertex v, i.e. $v \in V \setminus L$. According to the local shortest path tree for S_v, since $L \cap S_v \neq \emptyset$, we can find a child u of the root v in the local shortest path tree rooted at v such that $\text{dist}_G(v, L) > \text{dist}_G(u, L)$; in this case we set $\text{par}(v) = u$. For vertex $v \in L$, we can set $\text{par}(v) = v$. We can see now that pointers par denotes a rooted forest where the roots are sampled leaders. Furthermore, since $\forall v \notin L$, $(v, \text{par}(v))$ is from the local shortest path tree for S_v, we know that v and $\text{par}(v)$ are adjacent in the original graph G. After doing the above for all nodes v, the forest denoted by the resulting pointers par must be a subgraph of a spanning forest of G. We then apply the standard doubling algorithm to contract all the vertices to their leaders (roots), in $O(\log D)$ rounds. Therefore, the problem is reduced to finding a spanning forest in the contracted graph. The number of vertices remaining in the contracted graph is at most $\widetilde{O}(n/d)$, where $d = (m/n)^{\Theta(1)}$. By double-exponential speed problem size reduction technique, we can output a spanning forest in $O(\log D \cdot \log \log_{m/n} n)$ parallel time.

Although the above algorithm can output the edges of a spanning forest, it cannot output a rooted spanning forest. To output a rooted spanning forest, we follow a top-down construction. Suppose now we have a rooted spanning forest of the contracted graph. Since we have all the information of how vertices were contracted, we know contracted trees in the original graph. To

merge these contracted trees into the rooted spanning forest of the contracted graph, we only need to change the root of each contracted tree to a proper vertex in that tree. This changing root operation can be implemented by the doubling algorithm via a divide-and-conquer approach.

Since the spanning forest algorithm needs $O(\log \log_{m/n} n)$ phases to contract all vertices to a single vertex, the total parallel time to compute a rooted spanning forest is $O(\log D \cdot \log \log_{m/n} n)$. Furthermore, the depth of the rooted spanning forest will be at most $O(D^{O(\log \log_{m/n} n)})$. Thus, we can use the doubling algorithm to calculate the depth of the tree, and output this depth as an estimator of the diameter of the input graph.

4.2 Graph connectivity

In this section, we will discuss our graph connectivity algorithm in the sequential setting. We will see how to implement the algorithms efficiently in the parallel models in later sections.

4.2.1 Neighbor increment operation

In this section, we use truncated broadcasting procedure to increase the number of neighbors of every vertex and preserve the connectivity at the same time. The input of the procedure is an undirected graph $G = (V, E)$ and a parameter m which is larger than $|V|$. The output is a graph $G' = (V, E')$ such that for each vertex v, either the connected component which contains v is a clique or v has at least $\lceil (m/|V|)^{1/2} \rceil$ neighbors. Furthermore, $|E'| \leq |E| + m$. We use $\Gamma_G(v)$ to denote the neighbors of v in graph G, i.e. $\Gamma_G(v) = \{u \in V \mid \{u, v\} \in E\} \cup \{v\}$. Similarly, we let $\Gamma_{G'}(v)$ be the neighbors of v in G', i.e. $\Gamma_{G'}(v) = \{u \in V \mid \{u, v\} \in E'\} \cup \{v\}$. The procedure is shown in Algorithm 2.

The following definition defines the number of iterations of Algorithm 2.

Definition 4.2.1. *Given an undirected graph $G = (V, E)$ and a parameter $m \in \mathbb{Z}_{\geq 0}, m \geq 4|V|$, the number of iterations of* NEIGHBORINCREMENT(m, G) *(Algorithm 2) is the number of iterations t needed to run* TRUNCATEDBROADCASTING(G, b) *(Algorithm 1) in line 4.*

In the following lemma, we characterize the properties of Algorithm 2.

Algorithm 2 Neighbor Increment Operation

1: **procedure** NEIGHBORINCREMENT($m \geq 1, G = (V, E)$) ▷ Lemma 4.2.2.
2: Initially, $n \leftarrow |V|, b \leftarrow \lceil (m/n)^{1/2} \rceil$.
3: Regard G as a weighted graph $G = (V, E, w)$ with weights $w(u, v) = 1$ for every $\{u, v\} \in E$.
4: Run TRUNCATEDBROADCASTING(G, b), and let S_v be the output for $v \in V$. ▷ Algorithm 1.
5: Let $E' \leftarrow E \cup \bigcup_{v \in V} \{\{v, u\} \mid u \in S_v, u \neq v\}$.
6: Output $G' = (V, E')$.
7: **end procedure**

Lemma 4.2.2. *Let $G = (V, E)$ be an undirected graph, $m \in \mathbb{Z}_{\geq 0}$ which has $m \geq 4|V|$. Let $G' = (V, E')$ be the output of* NEIGHBORINCREMENT(m, G) *(Algorithm 2). We have:*

1. *The number of iterations (Definition 4.2.1) of* NEIGHBORINCREMENT(m, G) *is at most* $\min(\lceil \log(\mathrm{diam}(G)) \rceil, \lceil \log(m/n) \rceil) + 1$.

2. *For all $u, v \in V$, $\mathrm{dist}_G(u, v) < \infty \Leftrightarrow \mathrm{dist}_{G'}(u, v) < \infty$.*

3. *$\forall v \in V$, if $|\Gamma_{G'}(v)| < \lceil (m/n)^{1/2} \rceil$, then the connected component in G' which contains v is a clique. It also implies that $\forall u, v \in V$, if $|\Gamma_{G'}(v)| < \lceil (m/n)^{1/2} \rceil$ and $|\Gamma_{G'}(u)| \geq \lceil (m/n)^{1/2} \rceil$, then $\mathrm{dist}_{G'}(u, v) = \infty$.*

4. *$E \subseteq E', |E'| \leq |E| + m$.*

Proof. Consider property 1. According to Lemma 3.1.1, the number of iterations of TRUNCATEDBROADCASTING(G, b) (Algorithm 1) in line 4 is at most $\min(\lceil \log(\mathrm{diam}(G)) \rceil, \lceil \log(b) \rceil) + 1$. Since $b = \lceil (m/n)^{1/2} \rceil$, the number of iterations of NEIGHBORINCREMENT(m, G), is at most $\min(\lceil \log(\mathrm{diam}(G)) \rceil, \lceil \log(m/n) \rceil) + 1$.

For property 2, if u, v are in the same connected component in G, then since $E \subseteq E'$, u, v are in the same connected component in G'. If u, v are in the same connected component in G', then $\{u, v\} \in E, u \in S_v$ or $v \in S_u$. If $\{u, v\} \in E$, u, v are in the same connected component in G. Otherwise, according to Lemma 3.1.1, either u is a b-closest vertex of v in G or v is a b-closest vertex of u in G. In either case, u, v are in the same connected component in G.

Consider property 3. According to Lemma 3.1.1, S_v contains all b-closest vertices of v in G. Thus, if the number of neighbors of v in G' is less than b, S_v must contains all vertices which are

in the same connected component as v in G. Therefore, if the number of neighbors of v in G' is less than b, the connected component in G' which contains v must be a clique.

Now consider two vertices $u, v \in V$. Suppose the number of neighbors of v in G' is less than $b = \lceil (m/n)^{1/2} \rceil$, then we have that $\{p \in V \mid \mathrm{dist}_G(p, v) < \infty\}$ is a clique in G'. Thus, $\forall q \in \{p \in V \mid \mathrm{dist}_G(p, v) < \infty\}$, we have that the number of neighbors of q in G' is less than b. If the number of neighbors of u in G' is at least b, then $\mathrm{dist}_{G'}(u, v) = \infty$.

For property 4, we have $E \subseteq E'$ and $|E'| \leq |E| + \sum_{v \in V} |S_v| \leq |E| + n \cdot \lceil (m/n)^{1/2} \rceil \leq |E| + m$ where the second inequality follows from Lemma 3.1.1 and the choice of b. □

4.2.2 Random leader selection

Given an undirected graph $G = (V, E)$, to design a connected component algorithm, a natural way is constantly contracting the vertices in the same component. One way to do the contraction is that we randomly choose some vertices as leaders, then contract non-leader vertices to the neighbor leader vertices.

In this section, we show that if $\forall v \in V$, the number of neighbors of v is large enough, then we can just sample a small number of leaders such that for each non-leader vertex $v \in V$, there is at least one neighbor of v which is chosen as a leader. A more generalized statement is stated in the following lemma.

Lemma 4.2.3. *Let V be a vertex set with n vertices. Let $0 < \gamma \leq n, \delta \in (0, 1)$. For each $v \in V$, let S_v be an arbitrary subset of V satisfying that the size of S_v is at least γ. Let $l : V \to \{0, 1\}$ be a random hash function such that $\forall v \in V, l(v)$ are i.i.d. Bernoulli random variables, i.e.*

$$l(v) = \begin{cases} 1 & \text{with probability } p; \\ 0 & \text{otherwise.} \end{cases}$$

If $p \geq \min((10 \log(2n/\delta))/\gamma, 1)$, then, with probability at least $1 - \delta$,

1. $\sum_{v \in V} l(v) \leq \frac{3}{2} pn$;

2. $\forall v \in V, \exists u \in S_v$ *such that* $l(u) = 1$.

Proof. For a fixed vertex $v \in V$, we have

$$\begin{aligned}
&\Pr\left(\sum_{u\in S_v}(\mathbf{E}(l(u)) - l(u)) > \frac{1}{2}\sum_{u\in S_v}\mathbf{E}(l(u))\right)\\
\leq\ &\exp\left(-\frac{\frac{1}{2}\left(\frac{1}{2}\sum_{u\in S_v}\mathbf{E}(l(u))\right)^2}{\sum_{u\in S_v}\mathbf{Var}(l(u)) + \frac{1}{3}\cdot 1\cdot\frac{1}{2}\sum_{u\in S_v}\mathbf{E}(l(u))}\right)\\
\leq\ &\exp\left(-\frac{\frac{1}{2}\left(\frac{1}{2}\sum_{u\in S_v}\mathbf{E}(l(u))\right)^2}{\sum_{u\in S_v}\mathbf{E}(l(u)) + \frac{1}{3}\cdot 1\cdot\frac{1}{2}\sum_{u\in S_v}\mathbf{E}(l(u))}\right)\\
=\ &\exp\left(-\frac{3}{28}\cdot\sum_{u\in S_v}\mathbf{E}(l(u))\right) = \exp\left(-\frac{3}{28}\cdot p\cdot|S_v|\right) \leq \frac{\delta}{2n},
\end{aligned}$$

where the first inequality follows from Bernstein inequality and $|l(u) - E(l(u))| \leq 1$, the second inequality follows from $\mathbf{Var}(l(u)) \leq \mathbf{E}(l^2(u)) = \mathbf{E}(l(u))$, and the last inequality follows from $|S_v| \geq \gamma$, and $p \geq \min((10\log(2n/\delta))/\gamma, 1)$. Since $\frac{1}{2}\sum_{u\in S_v}\mathbf{E}(l(u)) \geq 1$, with probability at least $1 - \delta/(2n)$, $\sum_{u\in S_v} l(v) \geq 1$. By taking union bound over all S_v, with probability at least $1 - \delta/2$, $\forall v \in V, \exists u \in S_v, l(u) = 1$.

Similarly, we have

$$\begin{aligned}
&\Pr\left(\sum_{u\in V}(l(u) - \mathbf{E}(l(u))) > \frac{1}{2}\sum_{u\in V}\mathbf{E}(l(u))\right)\\
\leq\ &\exp\left(-\frac{\frac{1}{2}\left(\frac{1}{2}\sum_{u\in V}\mathbf{E}(l(u))\right)^2}{\sum_{u\in V}\mathbf{Var}(l(u)) + \frac{1}{3}\cdot 1\cdot\frac{1}{2}\sum_{u\in V}\mathbf{E}(l(u))}\right)\\
\leq\ &\exp\left(-\frac{\frac{1}{2}\left(\frac{1}{2}\sum_{u\in V}\mathbf{E}(l(u))\right)^2}{\sum_{u\in V}\mathbf{E}(l(u)) + \frac{1}{3}\cdot 1\cdot\frac{1}{2}\sum_{u\in V}\mathbf{E}(l(u))}\right)\\
=\ &\exp\left(-\frac{3}{28}\cdot\sum_{u\in V}\mathbf{E}(l(u))\right)\\
=\ &\exp\left(-\frac{3}{28}\cdot p\cdot|V|\right) \leq \frac{\delta}{2n} \leq \frac{\delta}{2}.
\end{aligned}$$

Since $\sum_{u\in V}\mathbf{E}(l(u)) = p\cdot n$, with probability at least $1 - \delta/2$, $\sum_{u\in V} l(u) \leq 1.5pn$.

By taking union bound, with probability at least $1 - \delta, \sum_{u \in V} l(u) \leq 1.5pn$ and $\forall v \in V, \exists u \in S_v, l(u) = 1$. □

If the number of neighbors of each vertex is not large, then we can still have a constant fraction of vertices which can contract to a leader.

Lemma 4.2.4. *Let V be a vertex set with n vertices. For each $v \in V$, let S_v be an arbitrary subset of $V \setminus \{v\}$ with size at least 2. Let $l : V \to \{0, 1\}$ be a random hash function such that $\forall v \in V, l(v)$ are i.i.d. Bernoulli random variables, i.e.*

$$l(v) = \begin{cases} 1 & \text{with probability } \frac{1}{2}; \\ 0 & \text{otherwise.} \end{cases}$$

Let $L = \{v \in V \mid l(v) = 1\} \cup \{v \in V \mid \forall u \in S_v, l(u) = 0\}$. Then $\mathbf{E}(L) \leq 0.75n$.

Proof. For $v \in V$, $\Pr(l(v) = 1) = \frac{1}{2}$. Let $u \in S_v \setminus \{v\}$. Then $\Pr(\forall x \in S_v, l(x) = 0) \leq \Pr(l(v) = 0, l(u) = 0) = 0.25$. $\mathbf{E}(|L|) = \sum_{v \in V} \Pr(v \in L) \leq 0.75n$. □

4.2.3 Tree contraction operation

In this section, we introduce the contraction operation. Firstly, let us introduce the concept of the parent pointers which can define a rooted forest.

Definition 4.2.5 (Parent pointers)**.** *Given a set of vertices V, let* par $: V \to V$ *satisfy that $\forall v \in V, \exists i > 0$ such that* $\text{par}^{(i)}(v) = \text{par}^{(i+1)}(v)$*, where $\forall v \in V, j > 0,$* $\text{par}^{(j)}(v)$ *is defined as* $\text{par}(\text{par}^{(j-1)}(v))$*, and* $\text{par}^{(0)}(v) = v$*. Then, we call such* par *a set of parent pointers on V. For $v \in V$, if* $\text{par}(v) = v$*, then we say v is a root of* par *.* par *can have more than one root. The depth of $v \in V$,* $\text{dep}_{\text{par}}(v)$ *is the smallest $i \in \mathbb{Z}_{\geq 0}$ such that* $\text{par}^{(i)}(v) = \text{par}^{(i+1)}(v)$*. The root of $v \in V$,* $\text{par}^{(\infty)}(v)$ *is defined as* $\text{par}^{(\text{dep}_{\text{par}}(v))}(v)$*. The depth of* par, dep(par) *is defined as* $\max_{v \in V} \text{dep}_{\text{par}}(v)$*.*

It is easy to see that a set of parent pointers par on V formed a rooted forest on V. For a vertex $v \in V$, if $\text{par}(v) = v$, then v is a root in the forest. Otherwise $\text{par}(v)$ is the parent of v in the forest.

In the following, we define the union operation of several sets of parent pointers.

Definition 4.2.6 (Union of parent pointers)**.** *Let* $\mathrm{par}_1 : V_1 \to V_1, \mathrm{par}_2 : V_2 \to V_2, \cdots, \mathrm{par}_k : V_k \to V_k$ *be k sets of parent pointers on vertex sets* $V_1, V_2, \cdots, V_k$ *respectively, where* $\forall i \neq j \in [k], V_i \cap V_j = \emptyset$*. Then* $\mathrm{par} = \mathrm{par}_1 \cup \mathrm{par}_2 \cup \cdots \cup \mathrm{par}_k$ *is a set of parent pointers on the vertex set* $V_1 \cup V_2 \cup \cdots \cup V_k$ *such that* $\forall i \in [k], v \in V_i, \mathrm{par}(v) = \mathrm{par}_i(v)$.

Now we focus on the parent pointers which can preserve the connectivity of the graph.

Definition 4.2.7. *Given a graph* $G = (V, E)$ *and a set of parent pointers* par *on* V*, if* $\forall v \in V$*, we have* $\mathrm{dist}_G(v, \mathrm{par}(v)) < \infty$*, then* par *is compatible with* G*.*

It is easy to show the following fact:

Fact 4.2.8. *Given a graph* $G = (V, E)$ *and a set of parent pointers* par *which is compatible with* G*, then* $\forall u, v \in V$ *with* $\mathrm{par}^{(\infty)}(u) = \mathrm{par}^{(\infty)}(v)$*, we have* $\mathrm{dist}_G(u, v) < \infty$.

Proof. By the definition of compatible, $\forall v \in V, \mathrm{dist}_G(v, \mathrm{par}(v)) < \infty$. By induction, $\forall l \in \mathbb{Z}_{>0}, v \in V$, we have $\mathrm{dist}_G(v, \mathrm{par}^{(l)}(v)) \leq \mathrm{dist}_G(v, \mathrm{par}^{(l-1)}(v)) + \mathrm{dist}_G(\mathrm{par}^{(l-1)}(v), \mathrm{par}^{(l)}(v)) < \infty$. Thus, for any pair of vertices $u, v \in V$, if $\mathrm{par}^{(\infty)}(u) = \mathrm{par}^{(\infty)}(v)$, then $\mathrm{dist}_G(u, v) \leq \mathrm{dist}_G(u, \mathrm{par}^{(\infty)}(u)) + \mathrm{dist}_G(\mathrm{par}^{(\infty)}(v), v) < \infty$. □

In this section, we describe a procedure which can be used to reduce the number of vertices. The input of the procedure is an undirected graph $G = (V, E)$ and a set of parent pointers $\mathrm{par} : V \to V$, where par is compatible with G. The output of the procedure will be the root of each vertex in V and an undirected graph $G' = (V', E')$ which satisfies $V' = \{v \in V \mid \mathrm{par}(v) = v\}, E' = \{\{u, v\} \subseteq V' \mid u \neq v, \exists \{p, q\} \in E, \mathrm{par}^{(\infty)}(p) = u, \mathrm{par}^{(\infty)}(q) = v\}$. Notice that V' only contains all the roots in the forest induced by par, and $|E'| \leq |E|$.

Lemma 4.2.9. *Let* $G = (V, E)$ *be an undirected graph,* $\mathrm{par} : V \to V$ *be a set of parent pointers (See Definition 4.2.5). Then* TreeContraction(G, par) *(See Algorithm 3) will output* $(G', g^{(r)})$ *with* $r \leq \lceil \log \mathrm{dep}(\mathrm{par}) \rceil$ *satisfies the following properties:*

Algorithm 3 Tree Contraction Operation

1: **procedure** TREECONTRACTION($G = (V, E)$, par : $V \to V$) ▷ Lemma 4.2.9, Corollary 4.2.12.
2: ▷ Output: $G' = (V', E')$, $\text{par}^{(\infty)}(v)$ for all $v \in V$.
3: Initially, for each $v \in V$ let $g^{(0)}(v) \leftarrow \text{par}(v)$. Let $V' \leftarrow \emptyset, E' \leftarrow \emptyset$.
4: $l \leftarrow 0$.
5: **for** $\exists v \in V, \text{par}(g^{(l)}(v)) \neq g^{(l)}(v)$ **do**
6: $l \leftarrow l + 1$.
7: For each $v \in V$, compute $g^{(l)}(v) \leftarrow g^{(l-1)}(g^{(l-1)}(v))$. ▷ $g^{(l)}$ is $\text{par}^{(2^l)}$.
8: **end for**
9: $r \leftarrow l$. ▷ r is the number of iterations, and is used in the analysis.
10: For $v \in V$, if $\text{par}(v) = v$, let $V' \leftarrow V' \cup \{v\}$.
11: For $\{u, v\} \in E$, if $g^{(r)}(u) \neq g^{(r)}(v)$, let $E' \leftarrow E' \cup \{\{g^{(r)}(u), g^{(r)}(v)\}\}$.
▷ $\forall v \in V$, contract v to $\text{par}^{(\infty)}(v)$.
12: **return** $g^{(r)}(v)$ as $\text{par}^{(\infty)}(v)$ for all $v \in V$, and $G' = (V', E')$.
13: **end procedure**

1. $\forall v \in V, g^{(r)}(v) = \text{par}^{(\infty)}(v)$.

2. $V' = \{v \in V \mid \text{par}(v) = v\}$.

3. $E' = \{\{u, v\} \subseteq V' \mid u \neq v, \exists \{p, q\} \in E, \text{par}^{(\infty)}(p) = u, \text{par}^{(\infty)}(q) = v\}$.

Proof. One crucial observation is the following claim.

Claim 4.2.10. *$\forall l \in \{0, 1, \cdots, r\}, v \in V$, we have $g^{(l)}(v) = \text{par}^{(2^l)}(v)$.*

Proof. The proof is by induction. When $l = 0$, $\forall v \in V, g^{(0)}(v) = \text{par}(v) = \text{par}^{(1)}(v)$, the claim is true. Suppose for $l - 1$, we have $\forall v \in V, g^{(l-1)}(v) = \text{par}^{(2^{l-1})}(v)$, then $\forall v \in V, g^{(l)}(v) = g^{(l-1)}(g^{(l-1)}(v)) = \text{par}^{(2^{l-1})}(\text{par}^{(2^{l-1})}(v)) = \text{par}^{(2^l)}(v)$. So the claim is true. □

If $r > \lceil \log \text{dep}(\text{par}) \rceil$, then $r - 1 \geq \lceil \log \text{dep}(\text{par}) \rceil$. Due to claim 4.2.10, we have $\forall v \in V, g^{(r-1)}(v) = \text{par}^{(2^{r-1})}(v) = \text{par}^{(\infty)}(v)$. Due to the condition in line 5, the loop will stop when $l \leq r - 1$ which leads to a contradiction to line 9. Thus, at the end of the algorithm, r should be at most $\lceil \log \text{dep}(\text{par}) \rceil$.

Since we have $\forall v \in V, \text{par}(g^{(r)}(v)) = g^{(r)}(v)$ at the end of the Algorithm 3, $\forall v \in V, g^{(r)}(v)$ must be $\text{par}^{(\infty)}(v)$. Then due to line 10 and line 11, we have $V' = \{v \in V \mid \text{par}(v) = v\}, E' = \{\{u, v\} \subseteq V' \mid u \neq v, \exists \{p, q\} \in E, \text{par}^{(\infty)}(p) = u, \text{par}^{(\infty)}(q) = v\}$. □

Definition 4.2.11. *Let $G = (V, E)$ be an undirected graph,* $\mathrm{par} : V \to V$ *be a set of parent pointers (See Definition 4.2.5). Then the number of iteration of* TREECONTRACTION(G, par) *is defined as the value of r at the end of the procedure.*

Corollary 4.2.12 (Preserved connectivity and diameter)**.** *Let $G = (V, E)$ be an undirected graph,* $\mathrm{par} : V \to V$ *be a set of parent pointers (See Definition 4.2.5) which is compatible (See Definition 4.2.7) with G. Then at the end of the Algorithm 3, $r \leq \lceil \log \mathrm{dep}(\mathrm{par}) \rceil$ and the output $(G', g^{(r)})$ will satisfy the following properties:*

1. $\mathrm{diam}(G') \leq \mathrm{diam}(G)$.
2. $\forall u, v \in V, \mathrm{dist}_G(u, v) < \infty \Rightarrow \mathrm{dist}_{G'}(\mathrm{par}^{(\infty)}(u), \mathrm{par}^{(\infty)}(v)) < \infty$.
3. $\forall u, v \in V, \mathrm{dist}_G(u, v) < \infty \Leftarrow \mathrm{dist}_{G'}(\mathrm{par}^{(\infty)}(u), \mathrm{par}^{(\infty)}(v)) < \infty$.

Proof. By Lemma 4.2.9, we have $r \leq \lceil \log \mathrm{dep}(\mathrm{par}) \rceil$, $V' = \{v \in V \mid \mathrm{par}(v) = v\}$ and $E' = \{\{u, v\} \subseteq V' \mid u \neq v, \exists \{p, q\} \in E, \mathrm{par}^{(\infty)}(p) = u, \mathrm{par}^{(\infty)}(q) = v\}$.

For any two vertices $u, v \in V$ which are in the same connected component in G, then there should be a path $u = u_1 \to u_2 \to \cdots \to u_p = v$ in graph G. So $\forall i \in [p-1], \{u_i, u_{i+1}\} \in E$ which means that either $\mathrm{par}^{(\infty)}(u_i) = \mathrm{par}^{(\infty)}(u_{i+1})$ or $\{\mathrm{par}^{(\infty)}(u_i), \mathrm{par}^{(\infty)}(u_{i+1})\} \in E'$. Thus, $\mathrm{par}^{(\infty)}(u_1) \to \mathrm{par}^{(\infty)}(u_2) \to \cdots \to \mathrm{par}^{(\infty)}(u_p)$ is a valid path in G', and the length of this path in G' is at most p. Thus, the properties 1 and 2 are true.

For any two vertices $u, v \in V$ which are not in the same connected component in G, but there is a path $\mathrm{par}^{(\infty)}(u) = u'_1 \to u'_2 \to \cdots \to u'_p = \mathrm{par}^{(\infty)}(v)$ in G', then it means that there exists vertices $u_{1,1}, u_{1,2}, u_{2,1}, u_{2,2}, \cdots, u_{p,1}, u_{p,2} \in V$ which satisfies

(a) $\forall i \in [p-1], \{u_{i,2}, u_{i+1,1}\} \in E, \mathrm{par}^{(\infty)}(u_{i,2}) = u'_i, \mathrm{par}^{(\infty)}(u_{i+1,1}) = u'_{i+1}$.

(b) $u_{1,1} = u, u_{p,2} = v$.

(c) $\forall i \in [p], \mathrm{par}^{(\infty)}(u_{i,1}) = \mathrm{par}^{(\infty)}(u_{i,2})$. By Fact 4.2.8, we have $\mathrm{dist}_G(u_{i,1}, u_{i,2}) < \infty$.

Thus, there exists a path from u to v. This contradicts to that u, v are not in the same connected component. Therefore, property 3 is also true. □

4.2.4 Connectivity algorithm

In this section, we describe the algorithm for graph connectivity/connected components problem in the classic sequential computing setting. The input is an undirected graph $G = (V, E)$, a space/rounds trade-off parameter m, and the rounds parameter $r \leq |V|$. The output is a function $\text{col} : V \to V$ such that $\forall u, v \in V, \text{dist}_G(u, v) < \infty \Leftrightarrow \text{col}(u) = \text{col}(v)$.

The algorithm is described in Algorithm 4. The following theorem shows the correctness of Algorithm 4.

Algorithm 4 Graph Connectivity

procedure CONNECTIVITY($G = (V, E), m, r$) ▷ Theorem 4.2.13, Theorem 4.2.20.
  Output: FAIL or $\text{col} : V \to V$.
  $n \leftarrow |V|$, $\forall v \in V$, $h_0(v) \leftarrow$ null.
  $G_0 = (V_0, E_0) = G$, i.e. $V_0 = V, E_0 = E$. $n_0 = n$.
  for $i = 1 \to r$ **do**
    $\forall v \in V$, $h_i(v) \leftarrow$ null. ▷ $h_i(v)$ is the vertex that v contracts to.
    $G'_i = (V'_i, E'_i) \leftarrow$ NEIGHBORINCREMENT(m, G_{i-1}). ▷ Algorithm 2.
    Compute $V''_i \leftarrow \{v \in V'_i \mid |\Gamma_{G'_i}(v)| \geq \lceil (m/n_{i-1})^{1/2} \rceil\}$.
    Compute $E''_i \leftarrow \{\{u, v\} \in E_{i-1} \mid u, v \in V''_i\}$.
    $G''_i = (V''_i, E''_i)$. ▷ G''_i is obtained by removing all the small components of G_i.
    Let $\gamma_i \leftarrow \lceil (m/n_{i-1})^{1/2} \rceil, p_i \leftarrow \min((30 \log(n) + 100)/\gamma_i, 1/2)$.
    Let $l_i : V''_i \to \{0, 1\}$ be a random hash function such that $\forall v \in V''_i, l_i(v)$ are i.i.d. Bernoulli random variables, and $\Pr(l_i(v) = 1) = p_i$.
    Let $L_i \leftarrow \{v \in V''_i \mid l_i(v) = 1\} \cup \{v \in V''_i \mid \forall u \in \Gamma_{G'_i}(v), l_i(u) = 0\}$. ▷ L_i are leaders.
    $\forall v \in V''_i$ with $v \in L_i$, let $\text{par}_i(v) \leftarrow v$.
    $\forall v \in V''_i$ with $v \notin L_i$, let $\text{par}_i(v) \leftarrow \min_{u \in L_i \cap \Gamma_{G'_i}(v)} u$. ▷ Non-leader finds a leader.
    Let $(G_i = (V_i, E_i), g_i^{(r'_i)}) \leftarrow$ TREECONTRACTION(G''_i, par_i). ▷ Algorithm 3.
    $n_i \leftarrow |V_i|$.
    For each $v \in V'_i \setminus V''_i$, let $h_i(v) \leftarrow \min_{u \in \Gamma_{G'_i}(v)} u$. ▷ Contract small component to one vertex.
    For each $v \in V''_i \setminus V_i$, let $h_i(v) \leftarrow g_i^{(r'_i)}(v)$. ▷ Contract non-leader to leader.
    For each $v \in V$, if $h_{i-1}(v) \neq$ null, then let $h_i(v) \leftarrow h_{i-1}(v)$.
  end for
  If $n_r \neq 0$, return FAIL.
  $((\widehat{V}, \widehat{E}), \text{col}) =$ TREECONTRACTION(G, h_r). ▷ Algorithm 3.
  return col .
end procedure

Theorem 4.2.13 (Correctness of Algorithm 4)**.** *Let $G = (V, E)$ be an undirected graph, $m \geq 4|V|$, and $r \leq |V|$ be the rounds parameter. If* CONNECTIVITY(G, m, r) *(Algorithm 4) does not output*

FAIL, then $\forall u, v \in V$, *we have* $\text{dist}_G(u, v) < \infty \Leftrightarrow \text{col}(u) = \text{col}(v)$.

Proof. Firstly, we show that the input of line 16 is valid.

Claim 4.2.14. $\forall i \in [r]$, par_i *is a set of parent pointers on* V_i''*, (See Definition 4.2.5) and is compatible (See Definition 4.2.7) with* G_i''.

Proof. $\forall v \in V_i''$, if $v \in L_i$, then $\text{par}_i(v) = v$. For $v \in V_i'' \setminus L_i$, due to property 3 of Lemma 4.2.2, we have $\text{par}_i(v) \in V_i''$. Since $\text{par}_i(v) \in L_i$, we have $\text{par}_i(\text{par}_i(v)) = \text{par}_i(v)$. Thus, $\text{par}_i : V_i'' \to V_i''$ is a set of parent pointers on V_i''. Due to property 2 of Lemma 4.2.2 and $\text{dist}_{G_i'}(\text{par}_i(v), v) < \infty$, we know that $\text{dist}_{G_{i-1}}(\text{par}_i(v), v) < \infty$. Thus, $\text{dist}_{G_i''}(\text{par}_i(v), v) < \infty$. It implies that par_i is compatible with G_i''. □

The following claim shows that the number of the remaining vertices cannot increase after each round.

Claim 4.2.15. *If* CONNECTIVITY(G, m, r) *does not output FAIL, then* $\forall i \in [r], V_i \subseteq V_i'' \subseteq V_i' = V_{i-1}$.

Proof. Let $i \in [r]$. Due to Claim 4.2.14, the input of line 16 is valid. Then, we can apply property 2 of Lemma 4.2.9 to get $V_i \subseteq V_i''$. By the construction of V_i'' we have $V_i'' \subseteq V_i'$. Since the procedure NEIGHBORINCREMENT(m, G_{i-1}) (Algorithm 2) does not change the vertex set, we have $V_i' = V_{i-1}$. □

Now, we show that $\forall u, v \in V_i, \text{dist}_{G_i}(u, v) < \infty \Leftrightarrow \text{dist}_G(u, v) < \infty$.

Claim 4.2.16. *If* CONNECTIVITY(G, m, r) *does not output* FAIL*, then* $\forall i \in [r], \forall u, v \in V_i$, *we have* $\text{dist}_{G_i}(u, v) < \infty \Leftrightarrow \text{dist}_G(u, v) < \infty$.

Proof. The proof is by induction. Suppose $\forall u, v \in V_{i-1}, \text{dist}_{G_{i-1}}(u, v) < \infty \Leftrightarrow \text{dist}_G(u, v) < \infty$. $\forall w, z \in V_i$, according to Claim 4.2.15, $w, z \in V_i''$. By property 2,3 of Corollary 4.2.12, and property 2 of Lemma 4.2.9, $\text{dist}_{G_i}(w, z) < \infty \Leftrightarrow \text{dist}_{G_i''}(w, z) < \infty$. Due to property 2,3 of Lemma 4.2.2, there is no edge in E_{i-1} between V_i'' and $V_i' \setminus V_i''$. According to Claim 4.2.15,

$w, z \in V_{i-1}$. Thus, $\text{dist}_{G_i''}(w, z) < \infty \Leftrightarrow \text{dist}_{G_{i-1}}(w, z) < \infty$. By induction hypothesis, we have $\forall w, z \in V_i, \text{dist}_{G_i}(w, z) < \infty \Leftrightarrow \text{dist}_G(w, z)$. □

The following claim states that once a vertex $v \in V$ is contracted to an another vertex, it will never be operated.

Claim 4.2.17. *Suppose* CONNECTIVITY(G, m, r) *does not output* FAIL. $\forall i \in \{0, 1, \cdots, r\}$, $v \in V$, *we have* $h_i(v) = \text{null} \Leftrightarrow v \in V_i$. *Furthermore,* $\forall v \in V, \exists j \in [r]$ *such that* $h_0(v) = h_1(v) = \cdots = h_{j-1}(v) = \text{null}$ *and* $h_j(v) = h_{j+1}(v) = \cdots = h_r(v) \neq \text{null}, \text{dist}_G(v, h_r(v)) < \infty$.

Proof. When $i = 0, \forall v \in V$, $h_0(v) = \text{null}, v \in V_0 = V$. Suppose it is true that $\forall v \in V, h_{i-1}(v) = \text{null} \Leftrightarrow v \in V_{i-1}$. If $v \notin V_i$, according to Claim 4.2.15, there are three cases: $v \in V_i'' \setminus V_i, v \in V_i' \setminus V_i'', v \notin V_{i-1}$. In the first case, due to line 19, $h_i(v) \neq \text{null}$. In the second case, due to line 18, $h_i(v) \neq \text{null}$, In the third case, due to line 20, $h_i(v) \neq \text{null}$. If $h_i(v) = \text{null}$, then $h_i(v)$ cannot be updated by line 18, line 19 or line 20 which implies that $v \in V_{i-1}, v \notin V_i' \setminus V_i'', v \notin V_i'' \setminus V_i$. Thus, $v \in V_i$.

Since the procedure does not FAIL, we have $n_r = 0$ which means that $\forall v \in V, h_r(v) \neq \text{null}$. Notice that by line 20, if $h_{i-1}(v) \neq \text{null}$, then $h_i(v) = h_{i-1}(v)$. Thus, $\forall v \in V, \exists j \in [r]$ such that $h_0(v) = h_1(v) = \cdots = h_{j-1}(v) = \text{null}$ and $h_j(v) = h_{j+1}(v) = \cdots = h_r(v) \neq \text{null}$.

For $v \in V$, if $h_j(v) \neq \text{null}$ and $h_{j-1}(v) = \text{null}$, then $h_j(v)$ can only be updated by 18 or line 19. In both cases, $\text{dist}_{G_{j-1}}(v, h_j(v)) < \infty$. By Claim 4.2.16, we have that $\text{dist}_G(v, h_j(v)) < \infty$. □

In the following, we show that h_r is a rooted tree such that $\text{dist}_G(u, v) < \infty \Leftrightarrow u, v$ have the same root. Due to Claim 4.2.17, if CONNECTIVITY(G, m, r) does not output FAIL, then $n_r = 0$ which implies that $\forall v \in V, h_r(v) \neq \text{null}$. Thus, we can define $h_r^{(k)}(v)$ for $k \in \mathbb{Z}_{>0}$ as applying h_r on v k times. $\forall v \in V$, by Claim 4.2.17, let $j \in [r]$ satisfy that $h_j(v) \neq \text{null}$ and $h_{j-1}(v) = \text{null}$. If $h_j(v)$ is updated by line 19, then $h_j(h_j(v)) = \text{null}$. If $h_j(v)$ is updated by line 18, then $h_j(h_j(v)) = h_j(v)$. In both cases, h_j cannot create a cycle. Thus, we can define $h_r^{(\infty)}(v) = h_r^{(k)}(v)$ for some k which satisfies $h_r(h_r^{(k)}(v)) = h_r^{(k)}(v)$.

Claim 4.2.18. *Suppose* CONNECTIVITY(G, m, r) *does not output* FAIL. *Then* $\forall u, v \in V$, *we have* $\mathrm{dist}_G(u, v) < \infty \Leftrightarrow h_r^{(\infty)}(u) = h_r^{(\infty)}(v)$.

Proof. Let $u, v \in V$. By Claim 4.2.17, if $h_r^{\infty}(u) = h_r^{\infty}(v)$ we have $\mathrm{dist}_G(u, v) < \infty$.

If $\mathrm{dist}_G(u, v) < \infty$, then let $u' = h_r^{(\infty)}(u), v' = h_r^{(\infty)}(v)$. By Claim 4.2.17, $\mathrm{dist}_G(u', v') \leq \mathrm{dist}_G(u, u') + \mathrm{dist}_G(u, v) + \mathrm{dist}_G(v, v') < \infty$, and we can find $j \in [r]$ such that $h_j(u') \neq \text{null}, h_{j-1}(u') = \text{null}$. Without loss of generality, we can assume $h_{j-1}(v') = \text{null}$ (otherwise we can swap u' and v'). Due to Claim 4.2.17, $u', v' \in V_{i-1}$. Since $h_j(u') = h_r(u') = u'$, $h_j(u')$ can be only updated by line 18, and $u' \in V'_j \setminus V''_j$. Then due to property 3 of Lemma 4.2.9, v' should be in $\Gamma_{G'_i}(u) \cup \{u\}$. Since $h_j(v') = h_r(v') = v'$, we can conclude that $u' = v'$. □

If CONNECTIVITY(G, m, r) does not output FAIL, then in line 23, col is exactly $h_r^{(\infty)}$. By Claim 4.2.18, we have $\forall u, v \in V, \mathrm{dist}_G(u, v) < \infty \Leftrightarrow \mathrm{col}(u) = \mathrm{col}(v)$.

□

Now let us consider the number of iterations of Algorithm 4 and the success probability.

Definition 4.2.19 (Total iterations)**.** *Let* $G = (V, E)$ *be an undirected graph,* $\mathrm{poly}(n) \geq m > 4n$, *and* $r \leq n$ *be the rounds parameter where* n *is the number of vertices in* G. *The total number of iterations of* CONNECTIVITY(G, m, r) *(Algorithm 4) is defined as* $\sum_{i=1}^{r}(k_i + r'_i)$, *where* k_i *denotes the number of iterations (See Definition 4.2.1) of* NEIGHBORINCREMENT(m, G_{i-1}) *(see line 7), and* r'_i *denotes the number of iterations (See Definition 4.2.11) of* TREECONTRACTION(G''_i, par_i) *(see line 16).*

Theorem 4.2.20 (Success probability and total iterations)**.** *Let* $G = (V, E)$ *be an undirected graph,* $\mathrm{poly}(n) \geq m > 4n$, *and* $r \leq n$ *be the rounds parameter where* $n = |V|$. *Let* $c > 0$ *be a sufficiently large constant. If* $r \geq c \log\log_{m/n}(n)$, *then with probability at least* 0.98, CONNECTIVITY(G, m, r) *(Algorithm 4) will not return* FAIL. *If* CONNECTIVITY(G, m, r) *succeeds, let* k_i *denote the number of iterations (See Definition 4.2.1) of* NEIGHBORINCREMENT(m, G_{i-1}) *(see line 7), and let* r'_i *denote the number of iterations of (See Definition 4.2.11) of* TREECONTRACTION(G''_i, par_i) *(see line 16), then*

1. $\forall i \in [r], r'_i = 0$.

2. $\forall i \in [r]$, k_i *is at most* $\lceil \log(\mathrm{diam}(G)) \rceil + 1$.

3. *The number of iterations of line 23 is at most* $\lceil \log r \rceil$.

4. $\sum_{i=1}^{r} k_i \leq O(r \log(\mathrm{diam}(G)))$.

Let $c_1 > 0$ *be a sufficiently large constant. If* $m \geq c_1 n \log^4 n$*, then with probability at least* 0.99, $\sum_{i=1}^{r} k_i \leq O(\log(\mathrm{diam}(G)) \log \log_{\mathrm{diam}(G)}(n))$. *If* $m < c_1 n \log^4 n$*, then with probability at least* 0.98, $\sum_{i=1}^{r} k_i \leq O(\log(\mathrm{diam}(G)) \log \log_{\mathrm{diam}(G)}(n) + (\log \log(n))^2)$.

Proof. Suppose CONNECTIVITY(G, m, r) succeeds. Property 1 follows from $\forall v \in V''_i, \mathrm{par}_i(\mathrm{par}_i(v)) = \mathrm{par}_i(v)$ and Lemma 4.2.9. Property 2 follows from $\mathrm{diam}(G_r) \leq \mathrm{diam}(G''_r) \leq \mathrm{diam}(G'_r) \leq \mathrm{diam}(G_{r-1}) \leq \mathrm{diam}(G''_{r-1}) \leq \mathrm{diam}(G'_{r-1}) \leq \cdots \leq \mathrm{diam}(G_0) = \mathrm{diam}(G)$ and property 1 of Lemma 4.2.2. Property 3 follows by the depth of h_r is at most r and Lemma 4.2.9. Property 4 follows from property 2.

Now let us prove the success probability. Let $i \in [r]$. If $p_i < 0.5$, then we can apply Lemma 4.2.3 on vertex set V''_i, parameter γ_i, and hash function l_i. Notice that the set S_v in the statement of Lemma 4.2.3 is $\Gamma_{G'_i}(v)$ in the algorithm. Notice that $|V''_i| \leq n$. Then in the i^{th} round, if $p_i < 0.5$, then with probability at most $1/(100n^2)$, L_i will be $\{v \in V''_i \mid l_i(v) = 1\}$, and $n_i = |L_i| \leq 1.5 p_i n_{i-1}$. By taking union bound over all $i \in [r]$, we have that with probability at least 0.99, event $\mathcal{E}$ happens: for all $i \in [r]$, if $p_i < 0.5$, then $n_i \leq 1.5 p_i n_{i-1} \leq 0.75 n_{i-1}$. Suppose $\mathcal{E}$ happens. For $i \in [r], p_i = 0.5$, if we apply Lemma 4.2.4, then condition on n_{i-1}, we have $\mathbf{E}(n_i) \leq 0.75 n_{i-1}$. Thus, we know $\forall i \in [r], \mathbf{E}(n_i) \leq 0.75 \, \mathbf{E}(n_{i-1}) \leq 0.75^i n$.

Next, we discuss the case for $p_0 = 0.5$ and the case for $p_0 < 0.5$ separately.

If $p_0 = 0.5$, then $m \leq n \cdot (600 \log n)^4$. By Markov's inequality, when $i^* \geq 4 \log_{4/3}(6000 \log n)$, with probability at least 0.99, $n_{i^*} \leq n/(600 \log n)^4$ and thus $p_{i^*} < 0.5$. Condition on this event and $\mathcal{E}$, we have

$$n_r \leq \frac{\left(\frac{\left(\frac{n_{i^*}^{1.5}}{m^{0.5}} (45 \log n + 150) \right)^{1.5}}{m^{0.5}} (45 \log n + 150) \right)^{\cdots}}{\cdots} \qquad \text{(Apply } r' = r - i^* \text{ times)}$$

$$
\begin{aligned}
&= \frac{n_{i^*}^{1.5^{r'}}}{m^{1.5^{r'}-1}}(45\log n+150)^{2\cdot(1.5^{r'}-1)}\\
&= n_{i^*}/(m/n_{i^*})^{1.5^{r'}-1}\cdot(45\log n+150)^{2\cdot(1.5^{r'}-1)}\\
&\le n/\left(m/\left(n_{i^*}(45\log n+150)^2\right)\right)^{1.5^{r'}-1}\\
&\le n/\left(m/\left(n_{i^*}(45\log n+150)^2\right)\right)^{1.5^{r'/2}}\\
&\le n/(m/n)^{1.5^{r'/2}} \le \frac{1}{2},
\end{aligned}
$$

where the second inequality follows from $n_{i^*} \le n$, the third inequality follows from $r' \ge 5$, the forth inequality follows from $n_{i^*} \le n/(600\log n)^4$, and the last inequality follows from $r' \ge \frac{2}{\log 1.5}\log\log_{m/n}(2n)$. Since $4n \le m \le n\cdot(600\log n)^4, \log\log_{m/n} n = \Theta(\log\log n)$. Let $c > 0$ be a sufficiently large constant. Thus, when $r \ge c\log\log_{m/n} n \ge i^* + r' = 4\log(6000\log n)/\log(4/3) + \frac{2}{\log 1.5}\log\log_{m/n}(2n)$, with probability at least 0.98, CONNECTIVITY(G, m, r) will not fail.

Since property 1 of Lemma 4.2.2, we have $k_i \le O(\log(\min(m/n_{i-1}, \mathrm{diam}(G))))$. Thus,

$$
\begin{aligned}
&\sum_{i=1}^{r} k_i = \sum_{i=1}^{i^*} k_i + \sum_{i=i^*+1}^{r} k_i \le O\left((\log\log n)^2\right) + \sum_{i=i^*+1}^{r} k_i\\
&\le O\left((\log\log n)^2\right) + \sum_{i:i\ge i^*+1, m/n_{i-1}\le \mathrm{diam}(G)} k_i + \sum_{i:i\le r, m/n_{i-1}>\mathrm{diam}(G)} k_i\\
&\le O\left((\log\log n)^2\right) + O\left(\sum_{i=0}^{\lceil \log_{1.25}\log_2(\mathrm{diam}(G))\rceil} \log(2^{1.25^i})\right) + O\left(\sum_{i=0}^{\lceil \log_{1.25}\log_{\mathrm{diam}(G)}(m)\rceil} \log(\mathrm{diam}(G))\right)\\
&\le O\left((\log\log n)^2\right) + O(\log(\mathrm{diam}(G))) + O(\log(\mathrm{diam}(G))\log\log_{\mathrm{diam}(G)}(n))\\
&\le O(\log(\mathrm{diam}(G))\log\log_{\mathrm{diam}(G)}(n) + (\log\log(n))^2),
\end{aligned}
$$

where the first inequality follows from $i^* = O(\log\log n)$ and $\forall i \le [i^*], m/n_{i-1} \le \mathrm{poly}(\log n)$, the third inequality follows from $m/n_{i+1} \ge (m/n_i)^{1.5}/(45\log n + 150) \ge (m/n_i)^{1.25}$.

If $m > n\cdot(600\log n)^4$, then $\forall i \in \{0\}\cup[r-1]$, we have $p_i < 0.5$. Since $\mathcal{E}$ happens. We have:

$$
n_r \le \frac{\left(\frac{\left(\frac{n^{1.5}}{m^{0.5}}(45\log n+150)\right)^{1.5}}{m^{0.5}}(45\log n+150)\right)^{\cdots}}{\cdots} \qquad \text{(Apply } r \text{ times)}
$$

$$
\begin{aligned}
&= \frac{n^{1.5^r}}{m^{1.5^r-1}}(45\log n + 150)^{2\cdot(1.5^r-1)} \\
&= n/(m/n)^{1.5^r-1} \cdot (45\log n + 150)^{2\cdot(1.5^r-1)} \\
&= n/\left(m/\left(n(45\log n + 150)^2\right)\right)^{1.5^r-1} \\
&\le n/\left(m/\left(n(45\log n + 150)^2\right)\right)^{1.5^{r/2}} \\
&\le n/\left(m/\left(n(200\log n)^2\right)\right)^{1.5^{r/2}} \\
&\le \frac{1}{2},
\end{aligned}
$$

where the second inequality follows from $r \ge 5$, the third inequality follows from $45\log n + 150 \le 200\log n$, and the last inequality follows from

$$
r \ge c\log\log_{m/n} n \ge 2\log_{1.5}\log_{(m/n)^{1/2}} 2n \ge 2\log_{1.5}\log_{m/(n(200\log n)^2)} 2n
$$

for a sufficiently large constant $c > 0$.

By property 1 of Lemma 4.2.2, we have $k_i \le O(\log(\min(m/n_{i-1}, \mathrm{diam}(G))))$. Thus,

$$
\begin{aligned}
\sum_{i=1}^{r} k_i &\le \sum_{m/n_{i-1} \le \mathrm{diam}(G)} k_i + \sum_{m/n_{i-1} > \mathrm{diam}(G)} k_i \\
&\le O\left(\sum_{i=0}^{\lceil \log_{1.25}\log_2(\mathrm{diam}(G))\rceil} \log(2^{1.25^i})\right) + O\left(\sum_{i=0}^{\lceil \log_{1.25}\log_{\mathrm{diam}(G)}(m)\rceil} \log(\mathrm{diam}(G))\right) \\
&\le O(\log(\mathrm{diam}(G))) + O(\log(\mathrm{diam}(G))\log\log_{\mathrm{diam}(G)}(n)),
\end{aligned}
$$

where the first inequality follows from $m/n_{i+1} \ge (m/n_i)^{1.5}/(45\log n + 150) \ge (m/n_i)^{1.25}$.

Since n_r is an integer, n_r must be 0 when $n_r \le 1/2$. Let $c > 0$ be a sufficiently large constant. For all $m \ge 4n$, if $r \ge c\log\log_{m/n} n$ then CONNECTIVITY(G, m, r) will succeed with probability at least 0.98. □

4.3 Spanning forest

In this section, we will discuss our spanning forest algorithm in the sequential setting. We will see how to implement the algorithms efficiently in the parallel models in later sections.

4.3.1 Multiple local shortest path trees

In this section, we show a procedure which is a generalization of neighbor increment procedure shown in Section 4.2.1. The input of the procedure is an undirected graph $G = (V, E)$ and a parameter m which is larger than $|V| = n$. The output will be n local shortest path trees (See Definition 4.3.1 shown below) such that $\forall v \in V$, there is a shortest path tree with root v. Furthermore, each shortest path tree either is a spanning tree of its connected component or it contains exact $\left\lceil (m/|V|)^{1/4} \right\rceil$ vertices. The formal definition of a local shortest path tree is as the following:

Definition 4.3.1 (Local shortest path tree (LSPT))**.** *Let V' be a set of vertices, v be a vertex in V', and* $\mathrm{par} : V' \to V'$ *be a set of parent pointers (See Definition 4.2.5) on V' which satisfies that v is the only root of* par*. Let $T = (V', \mathrm{par})$. Given an undirected graph $G = (V, E)$, if $V' \subseteq V$ and $\forall u \in V' \setminus \{v\}, \{u, \mathrm{par}(u)\} \in E, \mathrm{dep}_{\mathrm{par}}(u) = \mathrm{dist}_G(u, v)$, then we say T is a local shortest path tree (LSPT) in G, and T has root v. The vertex set (V' in the above) in T is denoted as V_T. The set of parent pointers (*par *in the above) in T is denoted as* par_T*. For short,* $\mathrm{dep}_{\mathrm{par}_T}$ *is denoted as* dep_T*, and* $\mathrm{dep}(\mathrm{par}(T))$ *is denoted as* $\mathrm{dep}(T)$*.*

The algorithm is described in Algorithm 5. The high level idea is that we use the distance information computed by the truncated broadcasting procedure to decide the parent of each vertex. The details of the algorithm is described in Algorithm 5, and the guarantees of the algorithm is stated in the following lemma.

Lemma 4.3.2. *Let $G = (V, E)$ be an undirected graph, and m be a parameter which is at least $16|V|$. Let $\left(\{\widetilde{T}(v) \mid v \in V\}, \{\mathrm{dep}_{\widetilde{T}(v)} \mid v \in V\}\right) = \textsc{MultipleLargeTrees}(G, m)$. (Algorithm 5) Then, the output satisfies the following properties.*

Algorithm 5 Local Shortest Path Trees

procedure MULTIPLELARGETREES($G = (V, E), m$) ▷ Lemma 4.3.2, Lemma 4.3.4.
  Initially, $n \leftarrow |V|, b \leftarrow \lceil (m/n)^{1/4} \rceil$.
  Regard G as a weighted graph $G = (V, E, w)$ with weights $w(u, v) = 1$ for every $\{u, v\} \in E$.
  Run TRUNCATEDBROADCASTING(G, b), and let $S_v, \mathrm{dist}(v, u)$ be the corresponding output for $v \in V, u \in S_v$. ▷ Algorithm 1.
  for $v \in V$ **do**
    Initialize $\widetilde{T}(v) = (S_v, \mathrm{par}_{\widetilde{T}(v)}), \mathrm{dep}_{\widetilde{T}(v)} : S_v \to \mathbb{Z}_{\geq 0}$.
    For $u \in S_v$, let $\mathrm{dep}_{\widetilde{T}(v)}(u) \leftarrow \mathrm{dist}(v, u)$.
    Let $\mathrm{par}_{\widetilde{T}(v)}(v) \leftarrow v$.
    For $u \in S_v \setminus \{v\}$, find an arbitrary $x \in S_v$ such that $\{u, x\} \in E$ and $\mathrm{dist}(v, u) = \mathrm{dist}(v, x) + 1$, set $\mathrm{par}_{\widetilde{T}(v)}(u) \leftarrow x$.
  end for
  Return $\{\widetilde{T}(v) \mid v \in V\}, \{\mathrm{dep}_{\widetilde{T}(v)} \mid v \in V\}$.
end procedure

1. $\forall v \in V, \widetilde{T}(v)$ *is a LSPT (See Definition 4.3.1) with root* v*, and* $\mathrm{dep}_{\widetilde{T}(v)}$ *records the depth of every vertex in* $\widetilde{T}(v)$.

2. $\forall v \in V, u \in V_{\widetilde{T}(v)}, w \in V \setminus V_{\widetilde{T}(v)}$*, it satisfies* $\mathrm{dist}_G(v, u) \leq \mathrm{dist}_G(v, w)$.

3. $\forall v \in V$*, either* $|V_{\widetilde{T}(v)}| \geq \lceil (m/n)^{1/4} \rceil$ *or* $V_{\widetilde{T}(v)} = \{u \in V \mid \mathrm{dist}_G(u, v) < \infty\}$.

4. $\forall v \in V, |V_{\widetilde{T}(v)}| \leq \lfloor (m/n)^{1/2} \rfloor$.

Proof. Let us first consider property 1. According to Lemma 3.1.1, the vertex set of $\widetilde{T}(v)$ is S_v which is the set of all b-closest vertices of v in G. Consider $u \in S_v$. By the construction of $\mathrm{par}_{\widetilde{T}(v)}(u)$, we have $\mathrm{dist}(v, \mathrm{par}_{\widetilde{T}(v)}(u)) < \mathrm{dist}(v, u)$ and $\mathrm{par}_{\widetilde{T}(v)}(v) = v$. Therefore, $\mathrm{par}_{\widetilde{T}(v)}$ is a set of parent pointers on S_v. Since S_v is the set of all b-closest vertices of v in G, for each $u \in S_v \setminus \{v\}$, there must exist at least one vertex $x \in S_v$ such that $\{u, x\} \in E$ and $\mathrm{dist}(v, u) = \mathrm{dist}(v, x) + 1$. It implies that we can always set $\mathrm{par}_{\widetilde{T}(v)}(u)$ successfully for every $u \in S_v$. Next we show that the depth of u in $\widetilde{T}(v)$ is exactly the same as $\mathrm{dist}_G(v, u)$. The proof is by induction on $\mathrm{dist}_G(v, u)$. When $\mathrm{dist}_G(v, u) = 0$, we have $u = v$. Since $\mathrm{par}_{\widetilde{T}(v)}(v) = v$, the depth of v is $0 = \mathrm{dist}_G(v, v)$. Now suppose the claim is true for all $x \in S_v$ satisfying $\mathrm{dist}_G(v, x) \leq s$. Consider a vertex $u \in S_v$ satisfying $\mathrm{dist}_G(v, u) = s + 1$. By the construction of $\mathrm{par}_{\widetilde{T}(v)}(u)$, we know that $\mathrm{dist}_G(v, \mathrm{par}_{\widetilde{T}(v)}(u)) = s$. By induction hypothesis, the depth of $\mathrm{par}_{\widetilde{T}(v)}(u)$ in $\widetilde{T}(v)$ is also s which implies that the depth of u

in $\widetilde{T}(v)$ is $s+1 = \text{dist}_G(v,u)$. By our construction of $\text{dep}_{\widetilde{T}(v)}$, $\forall u \in S_v, \text{dep}_{\widetilde{T}(v)}(u) = \text{dist}(v,u)$. According to Lemma 3.1.1, we have $\text{dist}_G(v,u) = \text{dist}(v,u)$. Thus, $\text{dep}_{\widetilde{T}(v)}$ records the depth of every vertex in $\widetilde{T}(v)$.

Consider property 2. Notice that $V_{\widetilde{T}(v)} = S_v$. According to Lemma 3.1.1, S_v is exact the set of all b-closest vertices of v in G. Thus, $\forall u \in S_v, w \notin S_v, \text{dist}_G(v,u) \leq \text{dist}_G(v,w)$.

Property 3 and property 4 directly follow from that $V_{\widetilde{T}(v)} = S_v$ is the set of all b-closest vertices of v in G (see Lemma 3.1.1) and $b = \lceil (m/n)^{1/4} \rceil \leq \lfloor (m/n)^{1/2} \rfloor$. □

Definition 4.3.3. *Let graph $G = (V,E)$, and let m be a parameter which is at least $16|V|$. The number of iterations of $\left(\{\widetilde{T}(v) \mid v \in V\}, \{\text{dep}_{\widetilde{T}(v)} \mid v \in V\}\right) = \text{MULTIPLELARGETREES}(G,m)$ (Algorithm 5) is defined as the number of iterations t needed to run $\text{TRUNCATEDBROADCASTING}(G,b)$ (Algorithm 1) in line 4.*

Lemma 4.3.4 (Number of iterations of Algorithm 5)**.** *Let $G = (V,E)$ be an undirected graph, and let m be a parameter which is at least $16|V|$. The number of iterations (see Definition 4.3.3) of $\left(\{\widetilde{T}(v) \mid v \in V\}, \{\text{dep}_{\widetilde{T}(v)} \mid v \in V\}\right) = \text{MULTIPLELARGETREES}(G,m)$ (Algorithm 5) is at most $\min(\lceil \log(\text{diam}(G)) \rceil, \lceil \log(m/n) \rceil) + 1$.*

Proof. According to Lemma 3.1.1, the number of iterations of $\text{TRUNCATEDBROADCASTING}(G,b)$ (Algorithm 1) in line 4 is at most $\min(\lceil \log(\text{diam}(G)) \rceil, \lceil \log(b) \rceil) + 1$. Since $b = \lceil (m/n)^{1/4} \rceil$, the number of iterations of $\text{MULTIPLELARGETREES}(m,G)$, is at most $\min(\lceil \log(\text{diam}(G)) \rceil, \lceil \log(m/n) \rceil) + 1$. □

4.3.2 Path generation and root changing

In this section, we show a procedure which can output a path from a certain vertex to the root in a rooted tree. Then we show how to use the procedure to change the root of a rooted tree to a certain vertex in the tree. To output the vertex-root path, we have two stages. The first stage is using doubling method to compute the depth and the 2^ith (for all $i \in \{0,1,\cdots,\log(\text{dep})\}$) ancestor of each vertex. The second stage is using divide-and-conquer technique to split the path

Algorithm 6 Depth and Ancestors of Every Vertex

procedure FINDANCESTORS (par : $V \to V$) ▷ Lemma 4.3.6.
  For $v \in V$ let $g_0(v) = \text{par}(v)$. If $\text{par}(v) = v$, let $h_0(v) = 0$. Otherwise, let $h_0(v) = \text{null}$.
  Let $l = 0$.
  for $\exists v \in V, h_l(v) = \text{null}$ **do**
    $l \leftarrow l + 1$.
    for $v \in V$ **do**
      Let $g_l(v) = g_{l-1}(g_{l-1}(v))$. ▷ g_l is $\text{par}^{(2^l)}$.
      if $h_{l-1}(v) \neq \text{null}$ **then** $h_l(v) = h_{l-1}(v)$.
      else if $h_{l-1}(g_{l-1}(v)) \neq \text{null}$ **then** $h_l(v) = h_{l-1}(g_{l-1}(v)) + 2^{l-1}$.
      else $h_l(v) = \text{null}$.
      end if
    end for
  end for
  Let $r = l, \text{dep}_{\text{par}} \leftarrow h_r$.
  return $r, \text{dep}_{\text{par}}, \{g_i : V \to V \mid i \in \{0\} \cup [r]\}$. ▷ $\text{dep}_{\text{par}} : V \to \mathbb{Z}_{\geq 0}$.
end procedure

into segments, and recursively find the path for each segment. Once we have the procedure to find the vertex-root path, then we can use it to implement root-changing. The idea is very simple, if we want to change the root to a certain vertex, we just need to find the path from that vertex to the root, and reverse the parent pointers of every vertex on the path. The path finding procedure is described in Algorithm 7. The root changing procedure is described in Algorithm 8.

Definition 4.3.5. *Let* $\text{par} : V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V. *The number of iterations of* FINDANCESTORS(par) *is defined as the value of* r *at the end of the procedure.*

Lemma 4.3.6. *Let* $\text{par} : V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V. *Let* $(r, \text{dep}_{\text{par}}, \{g_i \mid i \in \{0\} \cup [r]\}) = \text{FINDANCESTORS}(\text{par})$ *(Algorithm 6). Then the number of iterations (see Definition 4.3.5)* r *should be at most* $\lceil \log(\text{dep}(\text{par}) + 1) \rceil$, $\text{dep}_{\text{par}} : V \to \mathbb{Z}_{\geq 0}$ *records the depth of every vertex in* V, *and* $\forall i \in \{0\} \cup [r], v \in V\ g_i(v) = \text{par}^{(2^i)}(v)$.

Proof. h_l and g_l will satisfies the properties in the following claim.

Claim 4.3.7. $\forall i \in \{0\} \cup [r], v \in V\ g_i(v) = \text{par}^{(2^i)}(v)$, *and if* $\text{dep}_{\text{par}}(v) \leq 2^i - 1$ *then* $h_i(v) = \text{dep}_{\text{par}}(v)$. *Otherwise* $\text{dep}_{\text{par}}(v) = \text{null}$.

Algorithm 7 Path in a Tree

procedure FINDPATH (par : $V \to V, q \in V$) ▷ Lemma 4.3.8
  Output: $\text{dep}_{\text{par}} : V \to \mathbb{Z}_{\geq 0}, P \subseteq V, w \in V \cup \{\text{null}\}$.
  $(r, \text{dep}_{\text{par}}, \{g_i \mid i \in \{0\} \cup [r]\}) = \text{FINDANCESTORS}(\text{par})$ ▷ Algorithm 6.
  Let $S_0 = \{(q, g_r(q))\}, k = \lceil \log(\text{dep}_{\text{par}}(q)) \rceil$. ▷ S_0 contains $(q,$ the root of $q)$.
  for $i = 1 \to k$ **do** ▷ S_i is a set of segments partitioned the path from q to the root of q.
    Let $S_i \leftarrow \emptyset$.
    for $(x, y) \in S_{i-1}$ **do**
      if $\text{dep}_{\text{par}}(x) - \text{dep}_{\text{par}}(y) > 2^{k-i}$ **then** $S_i \leftarrow S_i \cup \{(x, g_{k-i}(x)), (g_{k-i}(x), y)\}$.
      else $S_i \leftarrow S_i \cup \{(x, y)\}$.
      end if
    end for
  end for ▷ S_k only contains segments with length at most 1.
  Let $P \leftarrow \{q\}$
  for $(x, y) \in S_k$ **do**
    Let $P \leftarrow P \cup \{y\}$
  end for
  Find $w \in P$ with $\text{dep}_{\text{par}}(w) = 1$. If w does not exist, let $w \leftarrow$ null.
  return $(\text{dep}_{\text{par}}, P, w)$
end procedure

Proof. The proof is by induction. The claim is obviously true when $i = 0$. Suppose the claim is true for $i-1$. We have $g_i(v) = g_{i-1}(g_{i-1}(v)) = \text{par}^{(2^{i-1})}(\text{par}^{(2^{i-1})}(v)) = \text{par}^{(2^i)}(v)$. If $h_i(v) \neq \text{null}$, then there are two cases. In the first case, we have $h_i(v) = h_{i-1}(v)$. By induction we know $h_i(v) = \text{dep}_{\text{par}}(v)$. In the second case, we have $h_i(v) = h_{i-1}(g_{i-1}(v)) + 2^{i-1} = \text{dep}_{\text{par}}(\text{par}^{(2^{i-1})}(v)) + 2^{i-1}$. Notice that in this case $h_{i-1}(v) = \text{null}$, thus by the induction, $\text{dep}_{\text{par}}(v) \geq 2^{i-1}$. Therefore, $\text{dep}_{\text{par}}(v) = \text{dep}_{\text{par}}(\text{par}^{(2^{i-1})}(v)) + 2^{i-1} = h_i(v)$. If $h_i(v) = \text{null}$, then it means that $h_{i-1}(\text{par}^{(2^{i-1})}(v)) = \text{null}$ which implies that $\text{dep}_{\text{par}}(v) \geq 2^i$. □

Due to the above claim, we know that if $i \geq \lceil \log(\text{dep}_{\text{par}}(v) + 1) \rceil$ then $h_i(v) \neq \text{null}$. Thus, we have $r \leq \lceil \log(\text{dep}(\text{par}) + 1) \rceil$. Since the procedure returns h_r as dep_{par}, the returned dep_{par} is correct. □

Lemma 4.3.8. *Let* $\text{par} : V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V*. Let* q *be a vertex in* V*. Let* $(\text{dep}_{\text{par}}, P, w) = \text{FINDPATH}(\text{par}, q)$ *(Algorithm 7). Then* $\text{dep}_{\text{par}} : V \to \mathbb{Z}_{\geq 0}$ *records the depth of every vertex in* V *and* $P \subseteq V$ *is the set of all vertices on the path from* q *to*

the root of q, i.e. $P = \{v \in V \mid \exists k \geq 0, v = \text{par}^{(k)}(q)\}$. *If* $\text{dep}_{\text{par}}(q) \geq 1$, *then* $w = \text{par}^{(\text{dep}_{\text{par}}(q)-1)}(q)$. *Furthermore, k should be at most* $\lceil \log(\text{dep}(\text{par})) \rceil$.

Proof. By Lemma 4.3.6, since $(r, \text{dep}_{\text{par}}, \{g_i \mid i \in \{0\} \cup [r]\}) = \text{FINDANCESTORS}(\text{par})$, we know r should be at most $\lceil \log(\text{dep}(\text{par}) + 1) \rceil$, $\text{dep}_{\text{par}} : V \to \mathbb{Z}_{\geq 0}$ records the depth of every vertex in V, and $\forall i \in \{0\} \cup [r], v \in V\ g_i(v) = \text{par}^{(2^i)}(v)$. Thus $k = \lceil \log(\text{dep}_{\text{par}}(q)) \rceil \leq \lceil \log(\text{dep}(\text{par}) + 1) \rceil$

Now let us prove that P is the vertex set of all the vertices on the path from q to the root of q. We use divide-and-conquer to get P. The following claim shows that S_i is a set of segments which is a partition of the path, and each segment has length at most 2^{k-i}.

Claim 4.3.9. $\forall i \in \{0\} \cup [k]$, S_i *satisfies the following properties:*

1. $\exists (x, y) \in S_i$ *such that* $x = q$.
2. $\exists (x, y) \in S_i$ *such that* $y = g_r(q)$.
3. $\forall (x, y) \in S_i, \text{dep}_{\text{par}}(y) - \text{dep}_{\text{par}}(x) \leq 2^{k-i}$.
4. $\forall (x, y) \in S_i$, *if* $y \neq g_r(q)$, *then* $\exists (x', y') \in S_i, x' = y$.
5. $\forall (x, y) \in S_i, \exists j \in \mathbb{Z}_{\geq 0}, \text{par}^{(j)}(x) = y$.

Proof. Our proof is by induction. According to line 4, all the properties hold when $i = 0$. Suppose all the properties hold for $i - 1$. For property 1, by induction we know there exists $(x, y) \in S_{i-1}$ such that $x = q$. Then by line 8 and line 9, there must be an (x, y') in S_i. For property 2, by induction we know there exists $(x, y) \in S_{i-1}$ such that $y = g_r(q)$. Thus, there must be an (x', y) in S_i. For property 3, if (x, y) is added into S_i by line 9, then $\text{dep}_{\text{par}}(x) - \text{dep}_{\text{par}}(y) \leq 2^{k-i}$. Otherwise, in line 8, we have $\text{dep}_{\text{par}}(x) - \text{dep}_{\text{par}}(g_{k-i}(x)) \leq 2^{k-i}, \text{dep}_{\text{par}}(g_{k-i}(x)) - \text{dep}_{\text{par}}(y) \leq 2^{k-i+1} - 2^{k-i} = 2^{k-i}$. For property 4, if (x, y) is added into S_i by line 9, then by induction there is $(y, y') \in S_{i-1}$, and thus by line 9 and line 8, there must be $(y, y'') \in S_i$. Otherwise, in line 8 will generate two pairs $(x, g_{k-i}(x)), (g_{k-i}(x), y)$. For $(x, g_{k-i}(x))$, the property holds. For $(g_{k-i}(x), y)$, there must be $(y, y') \in S_{i-1}$ and thus there should be $(y, y'') \in S_i$. For property 5, since $g_{k-i}(x) = \text{par}^{(k-i)}(x)$, for all pairs generated by line 8 and line 9, the property holds. □

Algorithm 8 Root Changing

procedure RootChange(par : $V \to V, q \in V$) ▷ Lemma 4.3.10.
  Output: $\widehat{\text{par}} : V \to V$.
  $(\text{dep}_{\text{par}}, P, w) = \text{FindPath}(\text{par}, q)$. ▷ Algorithm 7.
  $\forall v \in V \setminus P$, let $\widehat{\text{par}}(v) = \text{par}(v)$.
  Let $\widehat{\text{par}}(q) = q$.
  Let $h : \{0\} \cup [\text{dep}_{\text{par}}(q)] \to P$ such that $\forall i \in \{0\} \cup [\text{dep}_{\text{par}}(q)], h(i) = x$ where $\text{dep}_{\text{par}}(x) = i$.
  for $v \in P \setminus \{q\}$ **do** ▷ Reverse par of all the vertices on the path from q to the root of q.
    Let $\widehat{\text{par}}(v) = h(\text{dep}_{\text{par}}(v) + 1)$.
  end for
  return $\widehat{\text{par}}$.
end procedure

By Claim 4.3.9, we know

$$S_k = \{(q, \text{par}(q)), (\text{par}(q), \text{par}^{(2)}(q)), (\text{par}^{(2)}(q), \text{par}^{(3)}(q)), \cdots, (\text{par}^{(\text{dep}_{\text{par}}(q)-1)}(q), \text{par}^{(\text{dep}_{\text{par}}(q))}(q))\}.$$

Thus, P is the set of all the vertices on the path from q to the root of q. And $w = \text{par}^{(\text{dep}_{\text{par}}(q)-1)}(q)$ when $\text{dep}_{\text{par}}(q) \geq 1$. □

Lemma 4.3.10. *Let* $\text{par} : V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V. *Let* q *be a vertex in* V. *Let* $\widehat{\text{par}} = \text{RootChange}(\text{par}, q)$ *(Algorithm 8). Then* $\widehat{\text{par}} : V \to V$ *is still a set of parent pointers (See Definition 4.2.5) on* V. $\forall v \in V$, *if* $\text{par}^{(\infty)}(v) = \text{par}^{(\infty)}(q)$ *then* $\widehat{\text{par}}^{(\infty)}(v) = q$. *Otherwise* $\widehat{\text{par}}^{(\infty)}(v) = \text{par}^{(\infty)}(v)$. $\forall u \neq v \in V, \text{par}(v) = u \Leftrightarrow$ *either* $\widehat{\text{par}}(v) = u$ *or* $\widehat{\text{par}}(u) = v$. *Furthermore,* $\text{dep}(\widehat{\text{par}}) \leq 2\,\text{dep}(\text{par})$.

Proof. For a vertex $v \in V$, if $\{u \mid i \in \mathbb{Z}_{\geq 0}, u = \text{par}^{(i)}(v)\} \cap P = \emptyset$, then we have $\forall i \in \mathbb{Z}_{\geq 0}, \text{par}^{(i)}(v) = \widehat{\text{par}}^{(i)}(v)$. According to Lemma 4.3.8, $P = \{u \in V \mid i \in \mathbb{Z}_{\geq 0}, \text{par}^{(i)}(q) = u\}$. Then for all $v \in P \setminus \{q\}$, if $\text{par}(u) = v$ then $\widehat{\text{par}}(v) = u$. Thus, $\forall u \in P, \widehat{\text{par}}^{(\infty)}(u) = q$. Let i^* be the smallest number such that $\text{par}^{(i^*)}(v) \in P$. Then $\widehat{\text{par}}^{(i^*)}(v) \in P$. Thus, $\widehat{\text{par}}^{(\infty)}(v) = \widehat{\text{par}}^{(\infty)}(\widehat{\text{par}}^{(i^*)}(v)) = q$. Furthermore, we have $\forall v \in V, \text{dep}(\widehat{\text{par}}) \leq \text{dep}(\text{par}) + \text{dep}_{\text{par}}(q) \leq 2\,\text{dep}(\text{par})$. □

4.3.3 Spanning forest expansion

In this section, we give the definition of spanning forest. If we are given a spanning forest of a contracted graph and spanning trees of each contracted component, then we show a procedure which can merge them to get a spanning forest of the original graph. Before go to the details, let us formally define the spanning forest.

Definition 4.3.11 (Rooted Spanning Forest)**.** *Let $G = (V, E)$ be an undirected graph. Let* par *: $V \to V$ be a set of parent pointers which is compatible (Definition 4.2.7) with G. If $\forall u, v \in V, \mathrm{dist}_G(u, v) < \infty \Rightarrow \mathrm{par}^{(\infty)}(u) = \mathrm{par}^{(\infty)}(v)$, and $\forall v \in V, \mathrm{par}(v) \neq v \Rightarrow \{v, \mathrm{par}(v)\} \in E$, then we call* par *a rooted spanning forest of G.*

The Algorithm 9 shows how to combine the spanning forest in the contracted graph with local spanning trees to get a spanning forest in the graph before contraction. Figure 4.1 shows an example.

Lemma 4.3.12. *Let $G_2 = (V_2, E_2)$ be an undirected graph. Let $\widetilde{\mathrm{par}} : V_2 \to V_2$ be a set of parent pointers (See Definition 4.2.5) which satisfies that $\forall v \in V_2$ with $\widetilde{\mathrm{par}}(v) \neq v$, $\{v, \widetilde{\mathrm{par}}(v)\}$ must be in E_2. Let $G_1 = (V_1, E_1)$ be an undirected graph satisfies $V_1 = \{v \in V_2 \mid \widetilde{\mathrm{par}}(v) = v\}, E_1 = \{\{u, v\} \subseteq V_1 \mid u \neq v, \exists \{x, y\} \in E_2, \widetilde{\mathrm{par}}^{(\infty)}(x) = u, \widetilde{\mathrm{par}}^{(\infty)}(y) = v\}$. Let* par *$: V_1 \to V_1$ be a rooted spanning forest (See Definition 4.3.11) of G_1. Let $f : V_1 \times V_1 \to \{\mathrm{null}\} \cup (V_2 \times V_2)$ satisfy the following property: for $u \neq v \in V_1$, if $\mathrm{par}(u) = v$, then $f(u, v) \in \{(x, y) \in V_2 \times V_2 \mid \{x, y\} \in E_2, \widetilde{\mathrm{par}}^{(\infty)}(x) = u, \widetilde{\mathrm{par}}^{(\infty)}(y) = v\}$, and $f(v, u) \in \{(x, y) \in V_2 \times V_2 \mid \{x, y\} \in E_2, \widetilde{\mathrm{par}}^{(\infty)}(x) = v, \widetilde{\mathrm{par}}^{(\infty)}(y) = u\}$. Let $\widehat{\mathrm{par}} = \textsc{ForestExpansion}(\mathrm{par}, \widetilde{\mathrm{par}}, f)$. Then $\widehat{\mathrm{par}} : V_2 \to V_2$ is a rooted spanning forest of G_2. In addition,* $\mathrm{dep}(\widehat{\mathrm{par}}) \leq (2 \cdot \mathrm{dep}(\widetilde{\mathrm{par}}) + 1)(\mathrm{dep}(\mathrm{par}) + 1)$.

Proof. Let $x, y \in V_2, u = \widetilde{\mathrm{par}}^{(\infty)}(x), v = \widetilde{\mathrm{par}}^{(\infty)}(y) \in V_1$ if $\mathrm{dist}_{G_2}(x, y) < \infty$, then since $E_1 = \{(u', v') \in V_1 \times V_1 \mid u' \neq v', \exists \{x', y'\} \in E_2, \widetilde{\mathrm{par}}^{(\infty)}(x') = u', \widetilde{\mathrm{par}}^{(\infty)}(y') = v'\}$, it must be true that $\mathrm{dist}_{G_1}(u, v) < \infty$. Since par is a spanning forest of G_1, we have $\mathrm{par}^{(\infty)}(u) = \mathrm{par}^{(\infty)}(v)$. It suffices to say $\forall x \in V_2, \widehat{\mathrm{par}}^{(\infty)}(x) = \mathrm{par}^{(\infty)}(\widetilde{\mathrm{par}}^{(\infty)}(x))$. We can prove it by induction on $\mathrm{dep}_{\mathrm{par}}(\widetilde{\mathrm{par}}^{(\infty)}(x))$. Let

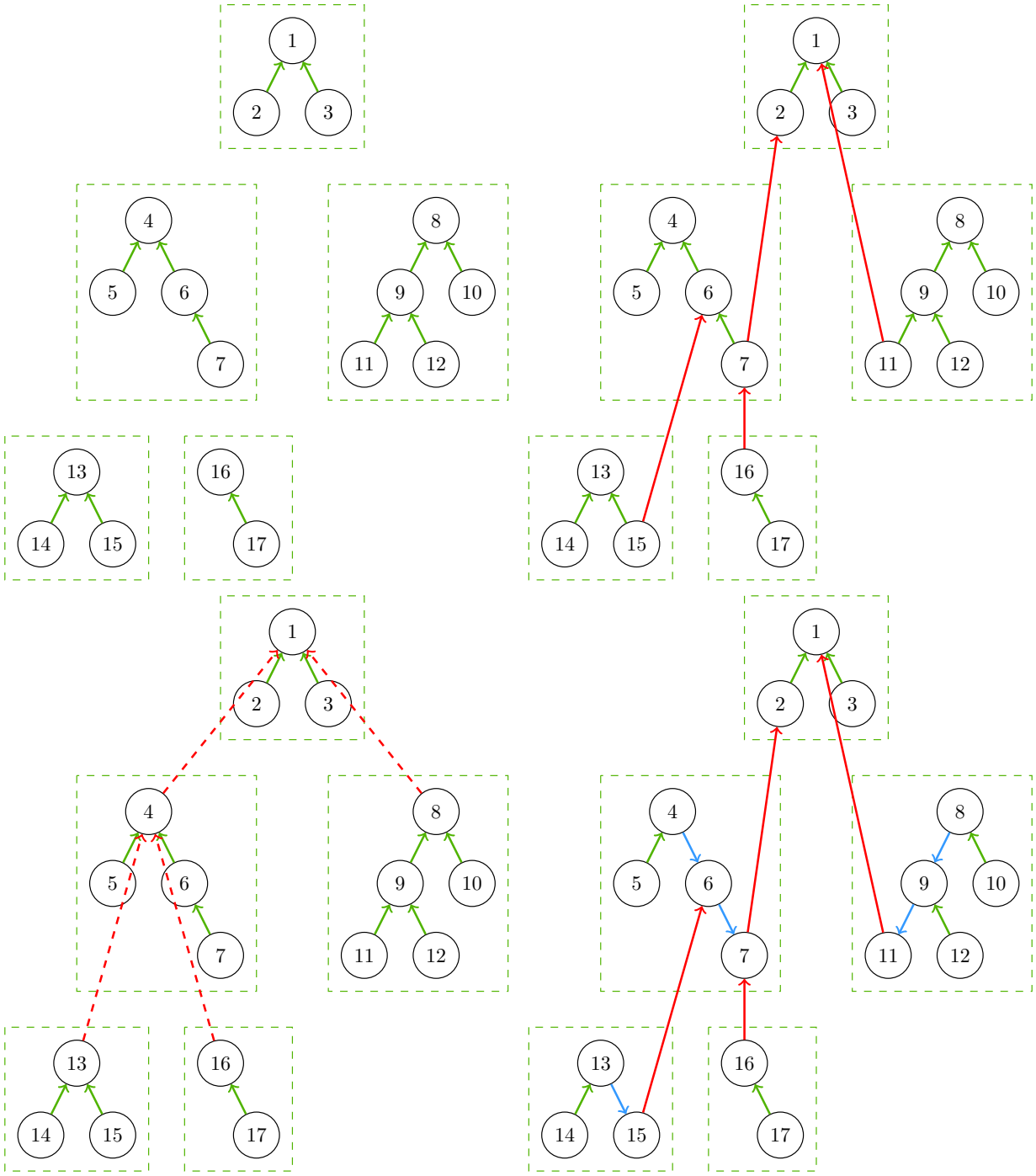

Figure 4.1: Each tree with green edges on the top-left is a rooted tree of each contracted component. For example, there are five components $\{1,2,3\}$, $\{4,5,6,7\}$, $\{8,9,10,11,12\}$, $\{13,14,15\}$, $\{16,17\}$. The dashed edges in the bottom-left figure is a root spanning tree of five components. The red edges in the top-right figure correspond to the dashed edges in the bottom-left figure before contraction. In bottom-right figure, by changing (see blue edges) the root of each contracted tree, we get a rooted spanning tree in the original graph

Algorithm 9 Spanning Forest Expansion

1: **procedure** FORESTEXPANSION(par : $V_1 \to V_1, \widetilde{\text{par}} : V_2 \to V_2, f : V_1 \times V_1 \to \{\text{null}\} \cup (V_2 \times V_2)$) ▷ Lemma 4.3.12.
2: Output: $\widehat{\text{par}} : V_2 \to V_2$.
3: $((V_2', \emptyset), \widetilde{\text{par}}^{(\infty)}) = \text{TREECONTRACTION}((V_2, \emptyset), \widetilde{\text{par}})$. ▷ Algorithm 3.
4: **for** $v \in V_1$ **do**
5: Let $V_2(v) = \{u \in V_2 \mid \widetilde{\text{par}}^{(\infty)}(u) = v\}$.
6: Let $\widetilde{\text{par}}_v : V_2(v) \to V_2(v)$ such that $\forall u \in V_2(v), \widetilde{\text{par}}_v(u) = \widetilde{\text{par}}(u)$.
7: **if** $\text{par}(v) \neq v$ **then**
8: Let $(x_v, y_v) = f(v, \text{par}(v))$.
9: $\widetilde{\text{par}}_v = \text{ROOTCHANGE}(\widetilde{\text{par}}_v, x_v)$. ▷ Algorithm 8.
10: Let $\widehat{\text{par}}(x_v) = y_v$, and $\forall u \in V_2(v) \setminus \{x_v\}, \widehat{\text{par}}(u) = \widetilde{\text{par}}_v(u)$.
11: **else** $\forall u \in V_2(v)$ let $\widehat{\text{par}}(u) = \widetilde{\text{par}}_v(u)$.
12: **end if**
13: **end for**
14: **return** $\widehat{\text{par}}$.
15: **end procedure**

$u = \widetilde{\text{par}}^{(\infty)}(x)$. If $\text{dep}_{\text{par}}(u) = 0$, then $\text{par}^{(\infty)}(u) = u$. In this case, we have $\widehat{\text{par}}^{(\infty)}(x) = \widetilde{\text{par}}^{(\infty)}(x) = u = \text{par}^{(\infty)}(u) = \text{par}^{(\infty)}(\widetilde{\text{par}}^{(\infty)}(x))$, and also we have $\text{dep}_{\widehat{\text{par}}}(x) = \text{dep}_{\widetilde{\text{par}}}(x)$. Now suppose for all $x \in V_2$ with $\text{dep}_{\text{par}}(\widetilde{\text{par}}^{(\infty)}(x)) \leq i - 1$, it has $\widehat{\text{par}}^{(\infty)}(x) = \text{par}^{(\infty)}(\widetilde{\text{par}}^{(\infty)}(x))$ and $\text{dep}_{\widehat{\text{par}}}(x) \leq i \cdot (2\,\text{dep}(\widetilde{\text{par}}) + 1)$. Let $y \in V_2$ satisfy $\text{dep}_{\text{par}}(\widetilde{\text{par}}^{(\infty)}(y)) = i$. Let $v = \widetilde{\text{par}}^{(\infty)}(y)$. By line 8 and the properties of f, we know $\widetilde{\text{par}}^{(\infty)}(x_v) = v$, and $\widetilde{\text{par}}^{(\infty)}(y_v) = \text{par}(v)$. By line 9, line 10 and Lemma 4.3.10, we have $\widehat{\text{par}}_v^{(\infty)}(y) = x_v, \widehat{\text{par}}(x_v) = y_v$. Thus, there must be $k \leq 2\,\text{dep}_{\widetilde{\text{par}}}(y)$ such that $\widehat{\text{par}}^{(k)}(y) = x_v$. Since $\widehat{\text{par}}^{(\infty)}(y_v) = \text{par}^{(\infty)}(v)$ and $\text{dep}_{\widehat{\text{par}}}(y_v) \leq i \cdot (2\,\text{dep}(\widetilde{\text{par}}) + 1)$, we have $\widehat{\text{par}}^{(\infty)}(y) = \text{par}^{(\infty)}(v) = \text{par}^{(\infty)}(\widetilde{\text{par}}^{(\infty)}(y))$ and $\text{dep}_{\widehat{\text{par}}}(y) \leq (i+1) \cdot (2\,\text{dep}(\widetilde{\text{par}}) + 1)$.

In addition, by the properties of f and Lemma 4.3.10, $\forall v \in V_2$ with $\widehat{\text{par}}(v) \neq v$, we have $\{v, \widehat{\text{par}}(v)\} \in E_2$. To conclude, $\widehat{\text{par}} : V_2 \to V_2$ is a spanning forest of G_2, and $\text{dep}(\widehat{\text{par}}) \leq (\text{dep}(\text{par}) + 1)(2\,\text{dep}(\widetilde{\text{par}}) + 1)$. □

4.3.4 Spanning forest algorithm

In this section, we show how to apply the ideas shown in connectivity algorithm to get an spanning forest algorithm. Algorithm 10 can output a spanning forest of a graph G, but the edges

are not orientated. Then in Algorithm 11, we assign each forest edge an direction thus it is a rooted spanning forest.

Before we prove the correctness of the algorithms, let us briefly introduce the meaning of each variables appeared in the algorithms.

In Algorithm 10, G_0 is the original input graph, for $i \in \{0\} \cup [r-1]$, G'_i is obtained by deleting all the small size connected components in G_i, and G_{i+1} is obtained by contracting some vertices of G'_i. For a vertex v in graph G_i, if $h_i(v) = \text{null}$, then it means that the connected component which contains v is deleted when obtaining G'_i. If $h_i(v) \neq \text{null}$, it means that the vertex v is contracted to the vertex $h_i(v)$ when obtaining G_{i+1}. par_i is a rooted forest (may not be spanning) in graph G_i, if a tree from the forest is spanning in G_i, then all the vertex in that tree will be deleted when obtaining G'_i. Otherwise all the vertices in that tree will be contracted to the root, and the root will be one of the vertex in G_{i+1}. Since each connected component in G_{i+1} is obtained by contraction of some vertices in a connected component in G_i, each edge in G_{i+1} must correspond to an edge in G_i where the end vertices of the edge are contracted to different vertices. Thus, each edge in G_i should correspond to an edge in G, and $g_i : E_i \to E$ records the such correspondence. D_i records the edges added to the spanning forest F in the i^{th} round. For each vertex v in graph G_i, $\widetilde{T}_i(v)$ is a local shortest path tree (See definition 4.3.1) which is either with a large size or is a spanning tree in the component of v. L_i is a set of random leaders in G'_i such that in each local shortest path tree $\widetilde{T}_i(v)$, there is at least one leader shown in the tree. The following lemmas formally state the properties of the algorithm.

Lemma 4.3.13. *Let $G = (V, E)$ be an undirected graph, m be a parameter which is at least $16|V|$, and r be a rounds parameter. If* SpanningForest(G, m, r) *(Algorithm 10) does not return FAIL, then* $\text{diam}(G) = \text{diam}(G_0) \geq \text{diam}(G'_0) \geq \text{diam}(G_1) \geq \text{diam}(G'_1) \geq \cdots \geq \text{diam}(G_r)$.

Proof. By property 3 of Lemma 4.3.2, $\forall [i] \in \{0\} \cup [r-1]$, there is no edge between $V_i \setminus V'_i$ and V'_i. Thus, $\text{diam}(G'_i) \leq \text{diam}(G_i)$. Then due to property 1 of Corollary 4.2.12, we have $\text{diam}(G_{i+1}) \leq \text{diam}(G'_i)$. □

Algorithm 10 Undirected Graph Spanning Forest

procedure SPANNINGFOREST($G = (V, E), m, r$) ▷ Corollary 4.3.17, Theorem 4.3.22.
  Output: FAIL or $\{V_i \subseteq V \mid i \in \{0\} \cup [r]\}, \{\mathrm{par}_i : V_i \to V_i \mid i \in \{0\} \cup [r-1]\}, \{h_i : V_i \to V_{i+1} \cup \{\mathrm{null}\} \mid i \in \{0\} \cup [r-1]\}, F \subseteq E$.
  $n_0 = n \leftarrow |V|, G_0 = (V_0, E_0) = (V, E)$.
  Let $g_0 : E_0 \to E$ be an identity map.
  Let $n'_0 = n_0$.
  for $i = 0 \to r-1$ **do**
    $D_i \leftarrow \emptyset$.
    $\left(\{\widetilde{T}_i(v) \mid v \in V_i\}, \{\mathrm{dep}_{\widetilde{T}_i(v)} \mid v \in V_i\}\right) \leftarrow$ MULTIPLELARGETREES(G_i, m). ▷ Algorithm 5.
    Let $V'_i \leftarrow \{v \in V_i \mid |V_{\widetilde{T}_i(v)}| \geq \lceil (m/n_i)^{1/4} \rceil\}, E'_i \leftarrow \{\{u, v\} \in E_i \mid u, v \in V'_i\}, G'_i = (V'_i, E'_i)$.
    $\forall v \in V_i \setminus V'_i$, let $h_i(v) \leftarrow \mathrm{null}, u_v \leftarrow \min_{u \in V_{\widetilde{T}_i(v)}} u$. Let $\mathrm{par}_i(v) \leftarrow \mathrm{par}_{\widetilde{T}_i(u_v)}(v)$.
    $\forall v \in V_i \setminus V'_i$, if $\mathrm{par}_i(v) \neq v$, then $D_i \leftarrow D_i \cup \{g_i(\mathrm{par}_i(v), v), g_i(v, \mathrm{par}_i(v))\}$.
    Let $\gamma_i \leftarrow \lceil (m/n_i)^{1/4} \rceil, p_i \leftarrow \min((30 \log(n) + 100)/\gamma_i, 1/2)$.
    Let $l_i : V'_i \to \{0, 1\}$ be chosen randomly s.t. $\forall v \in V'_i, l_i(v)$ are i.i.d. Bernoulli random variables with $\Pr(l_i(v) = 1) = p_i$.
    Let $L_i \leftarrow \{v \in V'_i \mid l_i(v) = 1\} \cup \{v \in V'_i \mid \forall u \in V_{\widetilde{T}_i(v)}, l_i(u) = 0\}$.
    For $v \in V'_i$, let $z_i(v) \leftarrow \arg\min_{u \in L_i \cap V_{\widetilde{T}_i(v)}} \mathrm{dep}_{\widetilde{T}_i(v)}(u)$. If $z_i(v) = v$, let $\mathrm{par}_i(v) \leftarrow v$.
    Otherwise, $(\mathrm{dep}_{\widetilde{T}_i(v)}, P_i(v), w_i(v)) \leftarrow$ FINDPATH($\mathrm{par}_{\widetilde{T}_i(v)}, z_i(v)$), and let $\mathrm{par}_i(v) \leftarrow w_i(v)$. ▷ Algorithm 7.
    Let $((V_{i+1}, E_{i+1}), \mathrm{par}_i^{(\infty)}) \leftarrow$ TREECONTRACTION($G'_i, \mathrm{par}_i : V'_i \to V'_i$). ▷ Algorithm 3.
    $G_{i+1} = (V_{i+1}, E_{i+1}), n_{i+1} \leftarrow |V_{i+1}|$.
    $\forall v \in V'_i, h_i(v) \leftarrow \mathrm{par}_i^{(\infty)}(v)$. If $\mathrm{par}_i(v) \neq v$, then $D_i \leftarrow D_i \cup \{g_i(\mathrm{par}_i(v), v), g_i(v, \mathrm{par}_i(v))\}$.
    Let $g_{i+1} : E_{i+1} \to E$ satisfy $g_{i+1}(u, v) \leftarrow \min_{\{x,y\} \in E_i, h_i(x) = u, h_i(y) = v} g_i(x, y)$.
    Let $n'_{i+1} \leftarrow n'_i + n_{i+1}$. If $n'_{i+1} > 40n$, then return FAIL.
  end for
  If $n_r \neq 0$, return FAIL.
  Let $F \leftarrow \bigcup_{i \in \{0\} \cup [r-1]} D_i$.
  return $\{V_i \mid i \in \{0\} \cup [r]\}, \{\mathrm{par}_i \mid i \in \{0\} \cup [r-1]\}, \{h_i \mid i \in \{0\} \cup [r-1]\}, F$.
end procedure

Lemma 4.3.14. *Let $G = (V, E)$ be an undirected graph, m be a parameter which is at least $16|V|$, and r be a rounds parameter. If* SPANNINGFOREST(G, m, r) *(Algorithm 10) does not return FAIL, then $\forall i \in \{0\} \cup [r-1]$, $\mathrm{dep}(\mathrm{par}_i) \leq \min(\mathrm{diam}(G), \lfloor (m/n_i)^{1/2} \rfloor)$.*

Proof. Let $v \in V_i$. If $v \in V_i \setminus V'_i$, then due to property 3 of Lemma 4.3.2, we have $V_{\widetilde{T}_i(v)} = V_{\widetilde{T}_i(u_v)}$. Due to Lemma 4.3.13 and Lemma 4.3.2, we have $\mathrm{dep}_{\mathrm{par}_i}(v) \leq \mathrm{dep}(\widetilde{T}_i(u_v)) \leq \min(\mathrm{diam}(G), \lfloor (m/n_i)^{1/2} \rfloor)$. For $v \in V_i$, we define $\mathrm{dist}_{G_i}(v, L_i) = \min_{u \in L_i} \mathrm{dist}_{G_i}(v, u)$. By Lemma 4.3.2, we know $\mathrm{dist}_{G_i}(v, L_i) = \mathrm{dist}_{G_i}(v, z_i(v))$. Since $\widetilde{T}_i(v)$ is a LSPT (See Definition 4.3.1), by applying Lemma 4.3.8, we know $\mathrm{dist}_{G_i}(v, L_i) = \mathrm{dist}_{G_i}(w_i(v), L_i) + 1$, and $(v, w_i(v)) \in E_i$. Thus, by induction on $\mathrm{dist}_{G_i}(v, L_i)$, we can

get $\mathrm{dep}_{\mathrm{par}_i}(v) \le \mathrm{dist}_{G_i}(v, L_i)$. By Lemma 4.3.13 and Lemma 4.3.2, we can conclude $\mathrm{dep}(\mathrm{par}_i)(v) \le \min(\mathrm{diam}(G), \lfloor (m/n_i)^{1/2} \rfloor)$. □

Lemma 4.3.15. *Let $G = (V, E)$ be an undirected graph, m be a parameter which is at least $16|V|$, and r be a rounds parameter. If* SpanningForest(G, m, r) *(Algorithm 10) does not return FAIL, then $\forall i \in \{-1, 0\} \cup [r-1]$, we can define*

$$\forall v \in V, h^{(i)}(v) = \begin{cases} v & i = -1; \\ h_i(h^{(i-1)}(v)) & h^{(i-1)}(v) \neq \text{null}; \\ \text{null} & \text{otherwise}. \end{cases}$$

Then we have following properties:

1. *If $h^{(i)}(v) \neq$ null, then $h^{(i)}(v) \in V_{i+1}$.*
2. *$\forall u, v \in V, h^{(i)}(u) \neq h^{(i)}(v), \{u, v\} \in E$, we have $\{h^{(i)}(u), h^{(i)}(v)\} \in E_{i+1}$.*
3. *$\forall \{x, y\} \in E_{i+1}, \{u, v\} = g_{i+1}(x, y)$, we have $\{u, v\} \in E, h^{(i)}(u) = x, h^{(i)}(v) = y$.*

Proof. For property 1, we can prove it by induction. It is true for $i = -1$. If $h_0(v) \neq$ null, we know $h_0(v)$ must be assigned at line 19. Due to property 2 of Lemma 4.2.9, $h_0(v) \in V_1$. Suppose $\forall v \in V, h^{(i-1)}(v) \neq$ null, we have $h^{(i-1)}(v) \in V_i$. For a vertex v with $h^{(i)}(v) \neq$ null, according to the definition of $h^{(i)}(v)$, we know $h^{(i-1)}(v) \neq$ null. Let $u = h^{(i-1)}(v)$. u must be a vertex in G_i by the induction hypothesis. Since $h^{(i)}(v) \neq$ null, we know $h_i(u) \neq$ null. Thus, $h_i(u)$ must be assigned at line 19. Due to property 2 of Lemma 4.2.9, $h_i(u)$ must be in G_{i+1}, which implies $h^{(i)}(v) \in V_{i+1}$.

For property 2, we can also prove it by induction. It is true for $i = -1$. If $\{u, v\} \in E$, then due to property 3 of Lemma 4.3.2, either both u, v are in V'_0 or both u, v are in $V_0 \setminus V'_0$. If both u, v are in $V_0 \setminus V'_0$, then $h_0(u) = h_0(v) =$ null. Otherwise, if $h_0(u) \neq h_0(v)$, then due to property 3 of Lemma 4.2.9, $\{h_0(u), h_0(v)\} \in E_1$. Now suppose we have $\forall u, v \in V$, if $h^{(i-1)}(u) \neq h^{(i-1)}(v), \{u, v\} \in E$, then $\{h^{(i-1)}(u), h^{(i-1)}(v)\} \in E_i$. Let $\{u, v\} \in E, h^{(i)}(u) \neq h^{(i)}(v)$. Let $x = h^{(i-1)}(u), y = h^{(i-1)}(v)$. Due to property 3 of Lemma 4.3.2, either both x, y are in V'_i or both are in $V_i \setminus V'_i$. If $x, y \in V_i \setminus V'_i$,

then $h_i(x) = h_i(y) = \text{null}$ which contradicts to $h^{(i)}(u) \neq h^{(i)}(v)$. Thus, both of $x, y \in V'_i$. Then due to property 3 of Lemma 4.2.9, $\{h_i(x), h_i(v)\} \in E_{i+1}$. Thus, $\{h^{(i)}(u), h^{(i)}(v)\} \in E_{i+1}$.

For property 3, we can prove it by induction. It is true for $i = -1$. Let us consider the case when $i = 0$. Due to property 3 of Lemma 4.2.9 and the definition of g_0, g_1, we have $\forall \{x, y\} \in E_1$, $\{u, v\} = g_1(x, y)$, $h_0(u) = x, h_0(v) = y, \{u, v\} \in E$. Now suppose the property holds for $i - 1$. Let $\{x, y\} \in E_{i+1}$. Then $g_{i+1}(x, y) = g_i(x', y')$ for some $\{x', y'\} \in E_i, h_i(x') = x, h_i(y') = y$. Let $\{u, v\} = g_i(x', y')$. By the induction hypothesis $\{u, v\} \in E, h^{(i-1)}(u) = x', h^{(i-1)}(v) = y'$. Thus, $h^{(i)}(u) = x, h^{(i)}(v) = y$. □

Lemma 4.3.16. *Let $G = (V, E)$ be an undirected graph, m be a parameter which is at least $16|V|$, and r be a rounds parameter. If* SpanningForest(G, m, r) *(Algorithm 10) does not return FAIL, then $\forall i \in \{-1, 0\} \cup [r - 1]$, we can define*

$$\forall v \in V, h^{(i)}(v) = \begin{cases} v & i = -1 \\ h_i(h^{(i-1)}(v)) & h^{(i-1)}(v) \neq \text{null} \\ \text{null} & \text{otherwise.} \end{cases}$$

Let $\forall i \in \{0\} \cup [r]$, $\widehat{G}_i = (V_i, \widehat{E}_i = \{\{x, y\} \mid \{u, v\} \in \bigcup_{j=i}^{r-1} D_j, h^{(j-1)}(u) = x, h^{(j-1)}(v) = y\})$. Then $\widehat{G}_i$ is a spanning forest of G_i.

Proof. The proof is by induction. When $i = r$, since $V_r = \emptyset$, $\widehat{G}_r = (\emptyset, \emptyset)$ is a spanning forest of G_r. Now suppose $\widehat{G}_{i+1}$ is a spanning forest of G_{i+1}. Let $u, v \in V_i$. By property 2, 3 of Lemma 4.3.15, we have $\widehat{E}_i \subseteq E_i$. Thus, if $\text{dist}_{G_i}(u, v) = \infty$, then $\text{dist}_{\widehat{G}_i}(u, v) = \infty$. If $\text{dist}_{G_i}(u, v) < \infty$, there are several cases:

1. If $h_i(u) = h_i(v) = \text{null}$, then due to line 10, we know $u_u = u_v$, and $\widetilde{T}_i(u_v)$ is a spanning tree of the component which contains u, v. Thus, $\widehat{G}_i$ has a spanning tree of the component which contains u, v.

2. If $h_i(u) = h_i(v) \neq \text{null}$, then $\text{par}_i : \{x \in V_i \mid h_i(x) = h_i(v)\} \to \{x \in V_i \mid h_i(x) = h_i(v)\}$ is

Algorithm 11 Rooted Spanning Forest

procedure ORIENTATE($\{V_i \mid i \in \{0\} \cup [r]\}, \{\mathrm{par}_i \mid i \in \{0\} \cup [r-1]\}, \{h_i \mid i \in \{0\} \cup [r-1]\}, F$)
▷ Takes the output of Algorithm 10 as input.
▷ Theorem 4.3.19
  Output: par : $V_0 \to V_0$.
  Let $F_0 = F$.
  for $i = 0 \to r - 1$ **do**
    Initialize $F_{i+1} \leftarrow \emptyset, f_{i+1} : V_{i+1} \times V_{i+1} \to \{\mathrm{null}\}$.
    $\forall \{u, v\} \in F_i, h_i(u) \neq h_i(v)$, let $F_{i+1} \leftarrow F_{i+1} \cup \{\{h_i(u), h_i(v)\}\}, f_{i+1}(h_i(u), h_i(v)) \leftarrow (u, v)$.
  end for
  $\widehat{\mathrm{par}}_r : \emptyset \to \emptyset$.
  for $i = r \to 1$ **do** ▷ $\widehat{\mathrm{par}}_i$ is the spanning forest of G_i.
    Let $\widetilde{V}_i \leftarrow V_i \cup \{v \in V_{i-1} \mid h_{i-1}(v) = \mathrm{null}, \mathrm{par}_{i-1}(v) = v\}$.
    Let $\widetilde{\mathrm{par}}_i : \widetilde{V}_i \to \widetilde{V}_i$ satisfy $\forall v \in V_i, \widetilde{\mathrm{par}}_i(v) = \widehat{\mathrm{par}}_i(v)$, and $\forall v \in \widetilde{V}_i \setminus V_i, \widetilde{\mathrm{par}}_i(v) = v$.
    Let $\widehat{\mathrm{par}}_{i-1} \leftarrow$ FORESTEXPANSION($\widetilde{\mathrm{par}}_i, \mathrm{par}_{i-1}, f_i$). ▷ Algorithm 9
  end for
  Return $\widehat{\mathrm{par}}_0$ as par .
end procedure

a tree, and $\forall y \in \{x \in V_i \mid h_i(x) = h_i(v)\}$, if $\mathrm{par}_i(y) \neq y$, then $\{y, \mathrm{par}_i(y)\} \in \widehat{E}_i$. Since $\widehat{G}_{i+1}$ does not have any cycle, there is a unique path from u to v in $\widehat{G}_i$.

3. If $h_i(u) \neq h_i(v)$, then neither of them can be null. Since $\widehat{G}_{i+1}$ is a spanning forest on G_{i+1}, there must be a unique path from $h_i(u)$ to $h_i(v)$ in $\widehat{G}_{i+1}$. Suppose the path in $\widehat{G}_{i+1}$ is $h_i(u) = p_1 - p_2 - \cdots - p_t = h_i(v)$. Then there must be a sequence of vertices in G_i, $u = p_{1,1}, p_{1,2}, p_{2,1}, p_{2,2}, \cdots, p_{t,1}, p_{t,2} = v$ such that $h_i(p_{j,1}) = h_i(p_{j,2}) = p_j$ and $\{p_{j-1,2}, p_{j,1}\} \in \widehat{E}_i$. Thus, there is a unique path from u to v.

Thus, $\widehat{G}_i$ is a spanning forest of G_i. □

Corollary 4.3.17 (Correctness of Algorithm 10)**.** *Let $G = (V, E)$ be an undirected graph, m be a parameter which is at least $16|V|$, and r be a rounds parameter. If* SPANNINGFOREST(G, m, r) *(Algorithm 10) does not return FAIL, then $\widehat{G}_0 = (V, F)$ is a spanning forest of G.*

Proof. Just apply Lemma 4.3.16 for $i = 0$ case. □

Lemma 4.3.18. *Let $G = (V, E)$ be an undirected graph, m be a parameter which is at least $16|V|$, and r be a rounds parameter. If* SpanningForest(G, m, r) *(Algorithm 10) does not return FAIL, then $\forall i \in \{-1, 0\} \cup [r-1]$, we can define*

$$\forall v \in V, h^{(i)}(v) = \begin{cases} v & i = -1 \\ h_i(h^{(i-1)}(v)) & h^{(i-1)}(v) \neq \text{null} \\ \text{null} & \text{otherwise.} \end{cases}$$

$\forall i \in \{0\} \cup [r-1], v \in V_i$ with $\text{par}_i(v) \neq v$, there exists $\{x, y\} \in F$ such that $h^{(i-1)}(x) = v, h^{(i-1)}(y) = \text{par}_i(v)$.

Proof. By line 11, line 19, $\forall i \in \{0\} \cup [r-1], v \in V_i, \text{par}_i(v) \neq v$, we have $g_i(v, \text{par}_i(v)), g_i(\text{par}_i(v), v) \in D_i \subseteq F$. Since $\{\text{par}_i(v), v\} \in E_i$, by property 3 of Lemma 4.3.15, $\{x, y\} = g_i(v, \text{par}_i(v))$ satisfies $h^{(i-1)}(x) = v, h^{(i-1)}(y) = \text{par}_i(v)$. □

Theorem 4.3.19 (Correctness of Algorithm 11)**.** *Let $G = (V, E)$ be an undirected graph, m be a parameter which is at least $16|V|$, and r be a rounds parameter. If* SpanningForest(G, m, r) *(Algorithm 10) does not return FAIL, then let the output be the input of* Orientate$(\cdot)$*, (Algorithm 11) and the output $\text{par} : V \to V$ of* Orientate$(\cdot)$ *will be a rooted spanning forest (See Definition 4.3.11) of G. Furthermore, $\text{dep}(\text{par}) \leq O(\text{diam}(G))^r$.*

Proof. The proof is by induction. We want to show $\widehat{\text{par}}_i$ is a rooted spanning forest of G_i. When $i = r$, since $V_r = \emptyset$, the claim is true. Now suppose we have $\widehat{\text{par}}_{i+1}$ is a spanning forest of G_{i+1}. Let $\widetilde{G}_{i+1} = (\widetilde{V}_{i+1}, E_{i+1})$. It is easy to see $\widetilde{\text{par}}_{i+1} : \widetilde{V}_{i+1} \to \widetilde{V}_{i+1}$ is a spanning forest of $\widetilde{G}_{i+1}$. An observation is $\widetilde{V}_{i+1} = \{v \in V_i \mid \text{par}_i(v) = v\}$. Thus, $\widetilde{\text{par}}_{i+1}, \text{par}_i$ satisfies the condition in Lemma 4.3.12 when invoking ForestExpansion$(\widetilde{\text{par}}_{i+1}, \text{par}_i, f_{i+1})$. By Lemma 4.3.18, we know f_{i+1} also satisfies the condition in Lemma 4.3.12 when we invoke ForestExpansion$(\widetilde{\text{par}}_{i+1}, \text{par}_i, f_{i+1})$. Thus, $\widehat{\text{par}}_i$ is a rooted spanning forest of G_i due to Lemma 4.3.12.

By Lemma 4.3.12, we have $\text{dep}(\widehat{\text{par}}_i) \leq 16 \, \text{dep}(\widehat{\text{par}}_{i+1}) \, \text{diam}(G)$. By induction, we have $\text{dep}(\text{par}) \leq O(\text{diam}(G))^r$. □

Lemma 4.3.20. *Let $G = (V, E)$ be an undirected graph, m be a parameter which is at least $16|V|$, and $r \leq n$ be a round parameter. If* SpanningForest(G, m, r) *(Algorithm 10) does not return FAIL, then with probability at least* 0.89, $\sum_{i=0}^{r} n_i \leq 40n$.

Proof. Since $V_r \subseteq V_{r-1} \subseteq \cdots \subseteq V_0 = V$, we have $n_r \leq n_{r-1} \leq n_{r-2} \leq \cdots \leq n$. Due to line 14, line 15 and line 17, we know $\forall i \in \{0\} \cup [r-1]$, $V_{i+1} = L_i$. If $p_i < 1/2$, we know $p_i = (30\log(n)+100)/\gamma_i$. Since $|V_{\widetilde{T}_i}(v)| \geq \gamma_i$, we can apply Lemma 4.2.3 to get $\Pr(|L_i| \leq 1.5p_i n_i) \geq \Pr(|L_i| \leq 0.75n_i) \geq 1 - 1/(100n)$. By taking union bound over all $i \in \{0\} \cup [r-1]$, with probability at least 0.99, if $p_i < 0.5$, then $n_{i+1} \leq 0.75n_i$. By applying Lemma 4.2.4, condition on n_i and $p_i = \frac{1}{2}$, we have $\mathbf{E}(n_{i+1}) \leq 0.75n_i$. By Markov's inequality, with probability at 0.89, we have $\sum_{i=0}^{r} n_i \leq 40n$. □

Now let us define the total iterations of Algorithm 10 as the following:

Definition 4.3.21 (Total iterations)**.** *Let graph $G = (V, E)$, $m \leq \text{poly}(|V|)$ be a parameter which is at least $16|V|$, and r be a rounds parameter. The total number of iterations of* SpanningForest(G, m, r) *(Algorithm 10) is defined as $\sum_{i=0}^{r-1}(k_i + k'_i)$, where $\forall i \in \{0\} \cup [r-1]$, k_i denotes the number of iterations (See Definition 4.3.3) of* MultipleLargeTrees(G_i, m) *(see line 8), and k'_i denotes the number of iterations (See Definition 4.2.11) of* TreeContraction(G'_i, par_i) *(see line 17).*

Theorem 4.3.22 (Success probability of Algorithm 10)**.** *Let $G = (V, E)$ be an undirected graph. Let $m \leq \text{poly}(n)$ and $m \geq 16|V|$. Let r be a rounds parameter. Let $c > 0$ be a sufficiently large constant. If $r \geq c \log\log_{m/n} n$, then with probability at least* 0.79, SpanningForest(G, m, r) *(Algorithm 10) does not return* FAIL*. Furthermore, let $\forall i \in \{0\} \cup [r-1]$, k_i be the number of iterations (See Definition 4.3.3) of* MultipleLargeTrees(G_i, m) *and k'_i be the number of iterations (See Definition 4.2.11) of* TreeContraction$(G'_i, \text{par}_i : V'_i \to V'_i)$*. Let $c_1 > 0$ be a sufficiently large constant. If $m \geq c_1 n \log^8 n$, then with probability at least* 0.99, $\sum_{i=0}^{r-1} k_i + k'_i \leq O(\min(\log(\text{diam}(G)) \log\log_{\text{diam}(G)}(n), r\log(\text{diam}(G))))$*. If $m < c_1 n \log^8 n$, then with probability at least* 0.98, $\sum_{i=0}^{r-1} k'_i + k_i \leq O(\min(\log(\text{diam}(G)) \log\log_{\text{diam}(G)}(n) + (\log\log n)^2, r\log(\text{diam}(G))))$.

Proof. Due to Lemma 4.3.20, with probability at last 0.89, we have $\forall i \in [r], n'_i \le 40n$. Thus, we can condition on that SPANNINGFOREST(G, m, r) will not fail on line 21.

Due to Lemma 4.3.4, $k_i \le O(\log(\mathrm{diam}(G_i))) \le O(\log(\mathrm{diam}(G)))$. Due to Corollary 4.2.12 and Lemma 4.3.14, $k'_i \le O(\log(\mathrm{diam}(G)))$. Thus, $\sum_{i\in\{0\}\cup[r-1]} k'_i + k_i \le O(r\log(\mathrm{diam}(G)))$.

Since $V_r \subseteq V_{r-1} \subseteq V_{r-2} \subseteq \cdots \subseteq V_0 = V$, we have $n_r \le n_{r-1} \le n_{r-2} \le \cdots \le n$. Due to line 14, line 15 and line 17, we know $\forall i \in \{0\} \cup [r-1]$, $V_{i+1} = L_i$. If $p_i < 1/2$, we know $p_i = (30\log(n)+100)/\gamma_i$. Since $|V_{\widetilde{T}_i}(v)| \ge \gamma_i$, we can apply Lemma 4.2.3 to get $\Pr(|L_i| \le 1.5p_in_i) \ge 1 - 1/(100n)$. By taking union bound over all $i \in \{0\} \cup [r-1]$, with probability at least 0.99, if $p_i < 0.5$, then $n_{i+1} \le 1.5p_in_i \le 0.75n_i$. Let $\mathcal{E}$ be the event that $\forall i \in \{0\} \cup [r-1]$, if $p_i < 0.5$, then $n_{i+1} \le 1.5p_in_i$. Now, we suppose $\mathcal{E}$ happens.

If $p_0 = 0.5$, then $m \le n \cdot (600\log n)^8$. By applying Lemma 4.2.4, $\mathbf{E}(n_{i+1}) = \mathbf{E}(|L_i|) \le 0.75\,\mathbf{E}(n_i) \le \cdots \le 0.75^{i+1}n$. By Markov's inequality, when $i^* \ge 8\log(6000\log n)/\log(4/3)$, with probability at least 0.99, $n_{i^*} \le n/(600\log n)^8$ and thus $p_{i^*} < 0.5$. Condition on this event and $\mathcal{E}$, we have

$$
\begin{aligned}
n_r &\le \cfrac{\left(\cfrac{\left(\frac{n_{i^*}^{1.25}}{m^{0.25}}(45\log n+150)\right)^{1.25}}{m^{0.25}}(45\log n+150)\right)^{\cdots}}{\cdots} && \text{(Apply } r' = r - i^* \text{ times)}\\
&= n_{i^*}/(m/n_{i^*})^{1.25^{r'}-1} \cdot (45\log n+150)^{4\cdot(1.25^{r'}-1)}\\
&\le n/\left(m/\left(n_{i^*}(45\log n+150)^4\right)\right)^{1.25^{r'}-1}\\
&\le n/\left(m/\left(n_{i^*}(45\log n+150)^4\right)\right)^{1.25^{r'/2}} \le n/(m/n)^{1.25^{r'/2}} \le 1/2,
\end{aligned}
$$

where the second inequality follows by $n_{i^*} \le n$, the third inequality follows by $r' \ge 5$, the forth inequality follows by $n_{i^*} \le n/(600\log n)^8$, and the last inequality follows by $r' \ge \frac{2}{\log 1.25}\log\log_{m/n}(2n)$. Since $16n \le m \le n\cdot(600\log n)^8$, $\log\log_{m/n} n = \Theta(\log\log n)$. Let $c > 0$ be a sufficiently large constant. Thus, when $r \ge c\log\log_{m/n} n \ge i^* + r' = 8\log(6000\log n)/\log(4/3) + \frac{2}{\log 1.25}\log\log_{m/n}(2n)$, with probability at least 0.98, $n_r = 0$ implies that SPANNINGFOREST(G, m, r) will not fail. Due to

Lemma 4.3.4, we have $k_i \le O(\min(\log(m/n_i), \log(\mathrm{diam}(G))))$. Thus,

$$
\begin{aligned}
&\sum_{i=0}^{r-1} k_i \\
&= \sum_{i=0}^{i^*} k_i + \sum_{i=i^*+1}^{r-1} k_i \\
&\le O\left((\log\log n)^2\right) + \sum_{i=i^*+1}^{r-1} k_i \\
&\le O\left((\log\log n)^2\right) + \sum_{i:i\ge i^*+1, m/n_i \le \mathrm{diam}(G)} k_i + \sum_{i:i\le r, m/n_i > \mathrm{diam}(G)} k_i \\
&\le O\left((\log\log n)^2\right) + O\left(\sum_{i=0}^{\lceil \log_{1.25}\log_2(\mathrm{diam}(G))\rceil} \log(2^{1.25^i})\right) + O\left(\sum_{i=0}^{\lceil \log_{1.25}\log_{\mathrm{diam}(G)}(m)\rceil} \log(\mathrm{diam}(G))\right) \\
&\le O\left((\log\log n)^2\right) + O(\log(\mathrm{diam}(G))) + O(\log(\mathrm{diam}(G))\log\log_{\mathrm{diam}(G)}(n)) \\
&\le O(\log(\mathrm{diam}(G))\log\log_{\mathrm{diam}(G)}(n) + (\log\log(n))^2),
\end{aligned}
$$

where the first inequality follows by $i^* = O(\log\log n)$ and $\forall i \le [i^*], m/n_i \le \mathrm{poly}(\log n)$, the third inequality follows by $m/n_{i+1} \ge (m/n_i)^{1.25}/(45\log n + 100) \ge (m/n_i)^{1.125}$. Due to Corollary 4.2.12 and Lemma 4.3.14, we also have $k_i' \le O(\min(\log(m/n_i), \log(\mathrm{diam}(G))))$. Then, by the same argument, we have $\sum_{i=0}^{r-1} k_i' = O(\log(\mathrm{diam}(G))\log\log_{\mathrm{diam}(G)}(n) + (\log\log(n))^2)$.

If $m > n \cdot (600\log n)^8$, then $\forall i \in \{0\} \cup [r-1]$, we have $p_i < 0.5$. Since $\mathcal{E}$ happens. We have:

$$
\begin{aligned}
n_r &\le \frac{\left(\frac{\left(\frac{n^{1.25}}{m^{0.25}}(45\log n+150)\right)^{1.25}}{m^{0.25}}(45\log n+150)\right)^{\cdots}}{\cdots} \qquad \text{(Apply } r \text{ times)} \\
&= \frac{n^{1.25^r}}{m^{1.25^r-1}}(45\log n+150)^{4\cdot(1.25^r-1)} \\
&= n/(m/n)^{1.25^r-1}\cdot(45\log n+150)^{4\cdot(1.25^r-1)} \\
&= n/\left(m/\left(n(45\log n+150)^4\right)\right)^{1.25^r-1} \\
&\le n/\left(m/\left(n(45\log n+150)^4\right)\right)^{1.25^{r/2}} \\
&\le n/\left(m/\left(n(200\log n)^4\right)\right)^{1.25^{r/2}}
\end{aligned}
$$

$$\leq \frac{1}{2},$$

where the second inequality follows by $r \geq 5$, the third inequality follows by $45 \log n + 150 \leq 200 \log n$, and the last inequality follows by

$$r \geq c \log \log_{m/n} n \geq 2 \log_{1.25} \log_{(m/n)^{1/2}} 2n \geq 2 \log_{1.25} \log_{m/(n(200 \log n)^4)} 2n,$$

for a sufficiently large constant $c > 0$. Since n_r is an integer, n_r must be 0 when $n_r \leq 1/2$. SPANNINGFOREST(G, m, r) will succeed with probability at least 0.99. Due to Lemma 4.3.4, we have $k_i \leq O(\min(\log(m/n_i), \log(\mathrm{diam}(G))))$. Thus,

$$\begin{aligned}
\sum_{i=0}^{r-1} k_i &\leq \sum_{m/n_i \leq \mathrm{diam}(G)} k_i + \sum_{m/n_i > \mathrm{diam}(G)} k_i \\
&\leq O\left(\sum_{i=0}^{\lceil \log_{1.25} \log_2(\mathrm{diam}(G)) \rceil} \log(2^{1.25^i}) \right) + O\left(\sum_{i=0}^{\lceil \log_{1.25} \log_{\mathrm{diam}(G)}(m) \rceil} \log(\mathrm{diam}(G)) \right) \\
&\leq O(\log(\mathrm{diam}(G))) + O(\log(\mathrm{diam}(G)) \log \log_{\mathrm{diam}(G)}(n)),
\end{aligned}$$

where the second inequality follows by $m/n_{i+1} \geq (m/n_i)^{1.25}/(45 \log n + 100) \geq (m/n_i)^{1.125}$. Due to Corollary 4.2.12 and Lemma 4.3.14, we also have $k'_i \leq O(\min(\log(m/n_i), \log(\mathrm{diam}(G))))$. Then, by the same argument, we have $\sum_{i=0}^{r-1} k'_i = O(\log(\mathrm{diam}(G))) + O(\log(\mathrm{diam}(G)) \log \log_{\mathrm{diam}(G)}(n))$.

□

4.4 Implementations in MPC model

In this section, we show how to implement our connectivity and spanning forest algorithms in the MPC model. For details of basic operations under MPC model, we refer readers to Section 2.3.

4.4.1 Neighbor increment operation

Lemma 4.4.1. *Let graph $G = (V, E), n = |V|, N = |V| + |E|$ and $m = \Theta(N^{1+\gamma})$ for some arbitrary $\gamma \in [0, 2]$.* NeighborIncrement(m, G) *(Algorithm 2) can be implemented in (γ, δ) −* MPC *model for any constant $\delta \in (0, 1)$. Furthermore, the parallel running time is linear in the number of iterations (see Definition 4.2.1) of* NeighborIncrement(m, G) *which is at most $O(\min(\log(\mathrm{diam}(G)), \log(m/n)))$.*

Proof. To implement line 3 of Algorithm 2, we can create a tuple $(\text{“}w\text{”}, (\{u, v\}, 1))$ for each tuple $(\text{“}E\text{”}, \{u, v\})$ stored in the system. According to Lemma 3.1.4, we can implement line 4 of Algorithm 2 in (γ, δ)-MPC model, and the parallel time is linear in the number of iterations (see Definition 4.2.1) of NeighborIncrement(m, G) which is at most $O(\min(\log(\mathrm{diam}(G)), \log(m/n)))$. To finally create E', we can create a tuple $(\text{“}E'\text{”}, \{u, v\})$ for each tuple $(\text{“}E\text{”}, \{u, v\})$, and create a tuple $(\text{“}E'\text{”}, \{x, y\})$ for each tuple $(\text{“}S_x\text{”}, y)$. Thus, all steps can be implemented in (γ, δ)-MPC model, and the overall parallel time is linear in the number of iterations (see Definition 4.2.1) of NeighborIncrement(m, G) which is at most $O(\min(\log(\mathrm{diam}(G)), \log(m/n)))$. □

4.4.2 Tree contraction operation

In this section, we show how to implement Algorithm 3 in MPC model.

Lemma 4.4.2. *Let graph $G = (V, E)$ and* par $: V \to V$ *be a set of parent pointers (see Definition 4.2.5) on the vertex set V.* TreeContraction$(G,$ par$)$ *(Algorithm 3) can be implemented in $(0, \delta)$ −* MPC *model for any constant $\delta \in (0, 1)$. Furthermore, the parallel running time is $O(r)$, where r is the number of iterations (see Definition 4.2.11) of* TreeContraction$(G,$ par$)$*, and $r = O(\mathrm{dep}(\mathrm{par}))$.*

Proof. Let $N = |V| + |E|$. Then the total space is $\Theta(N)$.

Initially, each machine scans its local memory. If there is a tuple $(\text{“}V\text{”}, v)$, then it queries the value of par(v). It needs $O(1)$ parallel time to answer all the queries (see **Multiple queries** in

Lemma 2.3.6). Then the machine creates a tuple ("$g^{(0)}$", $(v, \text{par}(v))$). Thus, in the initialization stage, mapping $g^{(0)}, \text{par}$, set V, E are stored in the system.

In the l^{th} iteration, Each machine scans its local memory. If there is a tuple ("V", v), then it queries the value of $g^{(l-1)}(v)$. This can be done by **Multiple queries**. Then it queries the value of $\text{par}(g^{(l-1)}(v))$. This can also be done by **Multiple queries**. If $\text{par}(g^{(l-1)}(v)) = g^{(l-1)}(v)$, it creates a tuple ("Done", v). Then the machines can compute the sizes (see Section 2.3.3) of V and Done. Each machine queries the size of V and Done. This can be done by **Multiple queries**. Then if $|V| = |\text{Done}|$, every machine knows that the iterations are finished. Otherwise, the machine which holds ("V", v) queries the value of $g^{(l-1)}(g^{(l-1)})(v)$. This can be done by **Multiple queries**. And then it creates a tuple ("$g^{(l)}$", $(v, g^{(l-1)}(g^{(l-1)})(v))$).

At the end, if a machine holds a tuple ("V", v), then it queries $\text{par}(v)$ (see **Multiple queries**). If $v = \text{par}(v)$, it creates a tuple ("V'", v). If a machine holds a tuple ("E", $\{u, v\}$), then it queries $g^{(r)}(u), g^{(r)}(v)$, and creates a tuple ("E'", $\{g^{(r)}(u), g^{(r)}(v)-\}$).

Since at the end of each iteration l, the system only stores mappings $\text{par} : V \to V, g^{(r)} : V \to V$, and sets V, E, the total space used is at most $O(N)$. Thus, we can implement the algorithm in $(0, \delta)$ − MPC model.

The total parallel time is $O(r)$. By Corollary 4.2.12, $r = O(\text{dep}(\text{par}))$. Thus, the total parallel time is $O(\text{dep}(\text{par}))$. □

4.4.3 Graph connectivity

Theorem 4.4.3. *Let graph $G = (V, E), n = |V|, N = |V| + |E|$ and $m = \Theta(N^{1+\gamma})$ for some arbitrary $\gamma \in [0, 2]$. Let $r > 0$ be a round parameter.* CONNECTIVITY(G, m, r) *(Algorithm 4) can be implemented in $(\gamma, \delta)-$*MPC *model for any constant $\delta \in (0, 1)$. Furthermore, the parallel running time is $O(R)$, where R is the total number of iterations (see Definition 4.2.19) of* CONNECTIVITY(G, m, r).

Proof. Initially, we store sets V_0, E_0, V, E and mapping h_0 in the system. Now consider the i^{th} round. Due to Lemma 4.4.1, line 7 can be implemented in total space $\Theta(m)$ and with $O(k_i)$ parallel time, where k_i is the number of iterations (See Definition 4.2.1) of NEIGHBORINCREMENT(m, G_{i-1}).

To store V_i' and E_i', we need total space $\Theta(m)$. Line 8 can be implemented by operations described in **Sizes of sets** and **Multiple queries** (see Section 2.3). Line 9 can be implemented by the operations described in **Set membership** and **Multiple queries**. To implement line 12, for each tuple $(\text{“}V_i''\text{”}, v)$, we can create a tuple $(\text{“}l_i\text{”}, (v, x))$ where $x = 1$ with probability p_i, $x = 0$ with probability $1 - p_i$. To calculate p_i, the machine only needs to know n_{i-1}. This can be done by the operations described in **Sizes of sets** and **Multiple queries**. Line 13 and line 14 can be implemented by operations described in **Set membership** and **Multiple queries**. For line 15, set $L_i \cap \Gamma_{G_i'}(v)$ can be computed by operations described in **Set membership** and **Multiple queries**. Then, by operations in **Indexing elements in sets** and **Multiple queries**, we can get $\min_{u \in L_i \cap \Gamma_{G_i'}(v)} u$. Finally, by operation described in **Multiple queries**, $\forall v \in V_i''$ with $v \notin L_i$, the tuple $(\text{“par}_i\text{”}, (v, x))$ can be created, where $x = \min_{u \in L_i \cap \Gamma_{G_i'}(v)} u$. Due to Lemma 4.4.2, line 16 can be implemented in total $\Theta(m)$ space and $O(r_i')$ parallel running time, where r_i' is the number of iterations (see Definition 4.2.11) of TreeContraction(G_i'', par_i). Line 18 can be implemented by operations in **Set membership**, **Indexing elements in sets** and **Multiple queries**. Line 19 can be implemented by operations in **Set membership** and **Multiple queries**. Line 20 can be implemented by **Multiple queries**. For other $v \in V$ with $h_i(v) = \text{null}$ assigned by line 6, we can use the operations in **Set membership** and **Multiple queries** to find those v, and create a tuple $(\text{“}h_i\text{”}, (v, \text{null}))$.

Thus, in the i^{th} round, the parallel time needed is $O(k_i + r_i')$. At the end of the i^{th} round, we only need to keep sets V_i, E_i, V, E and mapping h_i in the system. It will take total space at most $O(m)$.

Due to Lemma 4.4.2 and $\text{dep}(h_r) \leq r$, line 23 can be implemented in at most $O(m)$ total space and $O(\log r)$ parallel time.

Thus, the total parallel time is $O(\log r + \sum_{i=1}^{r}(k_i + r_i')) = O(\sum_{i=1}^{r}(k_i + r_i'))$. By definition 4.2.19, the total parallel time is $O(R)$, where R is the total number of iterations of $\textsc{Connectivity}(G, m, r)$. The total space in the computation is always at most $\Theta(m)$. □

Here, we are able to conclude the following theorem for graph connectivity problem.

Theorem 4.4.4. *For any $\gamma \in [0, 2]$ and any constant $\delta \in (0, 1)$, there is a randomized (γ, δ) – MPC algorithm (see Algorithm 4) which can output the connected components for any graph $G = (V, E)$*

in $O(\min(\log D \cdot \log(1/\gamma'), \log n))$ *parallel time, where* D *is the diameter of* $G, n = |V|, N = |V|+|E|$ *and* $\gamma' = (1+\gamma)\log_n \frac{2N}{n^{1/(1+\gamma)}}$. *The success probability is at least* 0.98. *In addition, if the algorithm fails, then it will return* FAIL.

Proof. The implementation of Algorithm 4 in MPC model is shown by Theorem 4.4.3. The correctness of Algorithm 4 is proved by Theorem 4.2.13. The total parallel time and the success probability of Algorithm 4 is proved by Theorem 4.2.20. □

4.4.4 Algorithms for local shortest path trees

In this section, we mainly explained how to implement local shortest path tree algorithms described in Section 4.3.1.

Lemma 4.4.5. *Let graph* $G = (V, E), n = |V|, N = |V| + |E|$ *and* $m = \Theta(N^{1+\gamma})$ *for some arbitrary* $\gamma \in [0, 2]$. MULTIPLELARGETREES(G, m) *(Algorithm 5) can be implemented in* (γ, δ) − MPC *model for any constant* $\delta \in (0, 1)$. *Furthermore, the parallel running time is* $O(r)$, *where* r *is the number of iterations (see Definition 4.3.3) of* MULTIPLELARGETREES(G, m), *and* $r = O(\min(\log(\mathrm{diam}(G)), \log(m/n)))$.

Proof. To implement line 3 of Algorithm 5, we can create a tuple $(\text{“}w\text{”}, (\{u, v\}, 1))$ for each tuple $(\text{“}E\text{”}, \{u, v\})$ stored in the system. According to Lemma 3.1.4, line 4 of Algorithm 5 can be implemented in (γ, δ)-MPC model, and the parallel time is linear in the number of iterations (see Definition 4.3.3) of MULTIPLELARGETREES(G, m) which is at most $O(\min(\log(\mathrm{diam}(G)), \log(m/n)))$. Next, we discuss how to implement the line 5-10 of Algorithm 5 for all $v \in V$ simultaneously using $O(1)$ parallel time and $O(m)$ total space. For each tuple $(\text{“}S_v\text{”}, u)$, create a tuple $(\text{“}V_{\widetilde{T}(v)}\text{”}, u)$. For each tuple $(\text{“}S_v\text{”}, u)$, query (see **Multiple queries**) $\mathrm{dist}(v, u)$ and create a tuple $(\text{“}\mathrm{dep}_{\widetilde{T}(v)}\text{”}, (u, \mathrm{dist}(v, u)))$. For each tuple $(\text{“}S_v\text{”}, u)$, query the index of u in S_v (see **Multiple queries** and **Indexing elements in sets**). Suppose the index is j, create a tuple $(\text{“}f_v\text{”}, (j, u))$. The function $f_v(j)$ denotes the j-th element in S_v. For each tuple $(\text{“}S_v\text{”}, u)$, query the size of S_v, and the system copy every S_v to $S_{v,1}, S_{v,2}, \cdots, S_{v,|S_v|}$ (see **Multiple queries**, **Sizes of sets**, and **Copies of sets**). For each tuple

("$S_{v,j}$", u), query $x = f_v(j)$ and check whether $\{x, u\} \in E$ (see **Multiple queries** and **Set membership**). If $\{x, u\} \in E$, query $\mathrm{dist}(v, u)$ and $\mathrm{dist}(v, x)$ and check whether $\mathrm{dist}(v, u) = \mathrm{dist}(v, x) + 1$ (see **Multiple queries**). If $\mathrm{dist}(v, u) = \mathrm{dist}(v, x) + 1$ is also satisfied, create a tuple ("$\mathrm{par}_{\widetilde{T}(v)}$", (u, x)). If for "$\mathrm{par}_{\widetilde{T}(v)}$" and u there are multiple tuples ("$\mathrm{par}_{\widetilde{T}(v)}$", (u, x_1)), ("$\mathrm{par}_{\widetilde{T}(v)}$", (u, x_2)), $\cdots$, only keep one arbitrary tuple (this can be done by sorting, see Theorem 2.3.1). It is easy to verify that above procedures only takes $O(1)$ parallel time. The bottleneck of the space usage is the operation **Copies of sets**. According to Lemma 4.3.2, the size of S_v is at most $O(\sqrt{m/n})$. Thus, the total space used by **Copies of sets** is at most $O(m)$ which implies that all operations can be implemented in (γ, δ)-MPC model. The overall parallel time is $O(r) = O(\min(\log(\mathrm{diam}(G)), \log(m/n)))$. □

4.4.5 Path generation and root changing

Lemma 4.4.6. *Let* par : $V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V. *Let* $n = |V|$. FINDANCESTORS(par) *(Algorithm 6) can be implemented in* (γ, δ) − MPC *model for any* $\gamma \geq \frac{\log\log(\mathrm{dep}(\mathrm{par}))}{\log n}$ *and any constant* $\delta \in (0, 1)$. *The parallel running time is* $O(r)$, *where* $r = O(\log(\mathrm{dep}(\mathrm{par})))$ *is the number of iterations (see Definition 4.3.5) of* FINDANCESTORS(par).

Proof. The structure of the whole algorithm is the same as the Algorithm 3 (see Lemma 4.4.2). All the steps can be done by operation described in **Multiple queries**.

Since the number of rounds needed is r, the parallel time is $O(r)$. For the total space, we need to store all the mappings $g_1, \cdots, g_r$. At the end of the i^{th} round, we need to store mapping h_i. According to Lemma 4.3.6, $r = O(\log(\mathrm{dep}(\mathrm{par})))$. Thus, the total space is $O(rn) = O(n \log(\mathrm{dep}(\mathrm{par})))$. □

Lemma 4.4.7. *Let* par : $V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V. *Let* q *be a vertex in* V, *and* $n = |V|$. FINDPATH(par, q) *(Algorithm 7) can be implemented in* (γ, δ) − MPC *model for any* $\gamma \geq \frac{\log\log n}{\log n}$ *and any constant* $\delta \in (0, 1)$. *The parallel running time is* $O(r)$, *where* r *is the number of iterations (see Definition 4.3.5) of* FINDANCESTORS(par) *(Algorithm 6).*

Proof. By Lemma 4.4.6, FINDANCESTORS(par) can be implemented in (γ, δ) – MPC model for $\gamma \geq \frac{\log\log n}{\log n}$ and any constant $\delta \in (0, 1)$. All the other other steps in the algorithm can be done by operation described in **Multiple queries**. Notice that, after each round, we need to do load balancing which can be done by operation described in **Load balance**.

The number of rounds must be smaller than $O(r)$, where r should be the number of iterations of FINDANCESTORS(par) according to Lemma 4.4.6.

We store all the mappings $g_i, \text{dep}_{\text{par}}$ in the system. They need $O(n \log n)$ total space. In the i^{th} round, we only need to additionally store set S_i which has size at most $O(n)$. Thus, the total space needed is at most $O(n \log n)$. □

Lemma 4.4.8. *Let* par : $V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V. *Let* q *be a vertex in* V. ROOTCHANGE(par, q) *(Algorithm 8) can be implemented in* (γ, δ) – MPC *model for any* $\gamma \geq \frac{\log\log n}{\log n}$ *and any constant* $\delta \in (0, 1)$. *The parallel running time is* $O(r)$, *where* r *is the number of iterations (see Definition 4.3.5) of* FINDANCESTORS(par) *(Algorithm 6).*

Proof. By Lemma 4.4.7, FINDPATH(par, q) can be implemented in (γ, δ) – MPC model. The remaining steps in the procedure can be implemented by the operation described by **Multiple queries**, and has $O(1)$ parallel running time.

The total space needed is the total space needed for FINDPATH(par, q) plus the space needed to store mapping $h, \widetilde{\text{par}}$. Thus the total space needed is $O(n \log n) + O(n) = O(n \log n)$.

The parallel running time is linear in the parallel running time of FINDPATH(par, q). Then, by Lemma 4.4.7, the parallel running time is $O(r)$ where r is the number of iterations (see Definition 4.3.5) of FINDANCESTORS(par). □

4.4.6 Spanning forest algorithm

Lemma 4.4.9. *Let* $G_2 = (V_2, E_2)$ *be an undirected graph. Let* $\widetilde{\text{par}} : V_2 \to V_2$ *be a set of parent pointers (See Definition 4.2.5) which satisfies that* $\forall v \in V_2$ *with* $\widetilde{\text{par}}(v) \neq v$, $(v, \widetilde{\text{par}}(v))$ *must be in* E_2. *Let* $G_1 = (V_1, E_1)$ *be an undirected graph satisfies* $V_1 = \{v \in V_2 \mid \widetilde{\text{par}}(v) = v\}, E_1 = \{\{u, v\} \subseteq V_1 \mid u \neq v, \exists \{x, y\} \in E_2, \widetilde{\text{par}}^{(\infty)}(x) = u, \widetilde{\text{par}}^{(\infty)}(y) = v\}$. *Let* par : $V_1 \to V_1$ *be a rooted spanning forest*

(See Definition 4.3.11) of G_1. *Let* $f : V_1 \times V_1 \to \{\text{null}\} \cup (V_2 \times V_2)$ *satisfy the following property: for* $u \neq v \in V_1$, *if* $\text{par}(u) = v$, *then* $f(u,v) \in \{(x,y) \in V_2 \times V_2 \mid \{x,y\} \in E_2, \widetilde{\text{par}}^{(\infty)}(x) = u, \widetilde{\text{par}}^{(\infty)}(y) = v\}$, *and* $f(v,u) \in \{(x,y) \in V_2 \times V_2 \mid \{x,y\} \in E_2, \widetilde{\text{par}}^{(\infty)}(x) = v, \widetilde{\text{par}}^{(\infty)}(y) = u\}$. *Let* $n = |V_2|$. *Then* ForestExpansion$(\text{par}, \widetilde{\text{par}}, f)$ *(Algorithm 9) can be implemented in* (γ, δ) − MPC *model for any* $\gamma \geq \log\log n / \log n$ *and any constant* $\delta \in (0,1)$ *in parallel running time* $O(R)$, *where* $R = \log(\text{dep}(\widetilde{\text{par}}))$.

Proof. Due to Lemma 4.4.2, line 3 can be done in $O(R)$ parallel time for $R = \log(\text{dep}(\widetilde{\text{par}}))$. Line 9 corresponds to multiple tasks, we can implement them parallelly by operations described in **Multiple queries**, and **Multiple Tasks** (see Section 2.3.6). By Lemma 4.4.8, the total space needed is at most $O(n \log n)$ and the parallel running time is at most $O(R)$ where $R = \log(\text{dep}(\widetilde{\text{par}}))$. □

Theorem 4.4.10. *Let graph* $G = (V,E), n = |V|, N = |V| + |E|$ *and* $m = \Theta(N^\gamma)$ *for some arbitrary* $\gamma \in [0,2]$. *Let* $r > 0$ *be a round parameter.* SpanningForest(G,m,r) *(Algorithm 10) can be implemented in* (γ, δ) − MPC *model for any constant* $\delta \in (0,1)$. *Furthermore, the parallel running time is* $O(R)$, *where* R *is the total number of iterations (see Definition 4.3.21) of* SpanningForest(G,m,r).

Proof. At the beginning of the algorithm, we just store sets V, E, V_0, E_0 and mapping g_0 in the system.

Consider the i^{th} round of the loop. By Lemma 4.4.5, line 8 can be implemented in total space $\Theta(m)$ and in parallel running time $O(k_i)$ where k_i is the number of iterations (see Definition 4.3.3) of MultipleLargeTrees(G_i, m). Line 9 can be implemented by operations described in **Sizes of sets**, **Set membership**, and **Multiple queries**. Line 10 can be implemented by operations described in **Indexing elements in sets**, **Set membership**, and **Multiple queries**. In line 12, to calculate γ_i, we need to query n_i, this can be done by operations described in **Sizes of sets** and **Multiple queries**. In line 14, to compute L_i, we only need operations described in **Set membership** and **Multiple queries**. Line 15 can be implemented by operations shown in **Set membership**, **Indexing elements in sets** and **Multiple queries**. By Lemma 4.4.7, for line 16, there are multiple

tasks each can be implemented in $O(|V_{\widetilde{T}_i(v)}| \log |V_{\widetilde{T}_i(v)}|)$ total space, and $O(k_i)$ parallel time. We can schedule these multiple tasks (see Section 2.3.6) such that we can finish them in parallel in $O(k_i)$ parallel time. According to Lemma 4.4.2, for line 17, we can implement it in $O(n_i) = O(n)$ total space, and in $O(k'_i)$ parallel time, where k'_i is the number of iterations (see Definition 4.2.11) of TREECONTRACTION(G'_i, par_i). Line 19 can be done by the operation described in **Multiple queries**. Line 20 can be done by the operation described in **Indexing elements in sets** and **Multiple queries**.

Thus, the parallel time is $O(R)$, where $R = \sum_{i=0}^{r-1}(k_i + k'_i)$. By definition of the total number of iterations (see Definition 4.3.21) of SPANNINGFOREST(G, m, r). R is the total number of iterations of SPANNINGFOREST(G, m, r).

For the space, we store all the sets V, E, V_i, D_i and mappings par_i, h_i in all the rounds. Notice that $\sum_{i=0}^{r} |V_i| \leq 40|V|$. Thus this part takes only $O(N)$ space. In the i^{th} round, we additionally store all the sets $V_{\widetilde{T}_i(v)}, V'_i, E'_i, L_i$ and all the mappings $\text{par}_{\widetilde{T}_i(v)}, \text{dep}_{\widetilde{T}_i(v)}, l_i, z_i$. The total space for this part is at most $O(m)$. For line 16, it creates multiple tasks. The input of each task is at most $|V_{\widetilde{T}_i(v)}| \leq (m/n_i)^{1/2}$. There are at most n_i tasks, and by Lemma 4.4.7, each task will need space at most $O(|V_{\widetilde{T}_i(v)}| \log |V_{\widetilde{T}_i(v)}|)$. Thus, the space for this part is at most $O(m)$. To conclude, the total space needed is at most $O(m)$.

□

Theorem 4.4.11. *Let graph $G = (V, E), n = |V|, N = |V| + |E|$ and $m = \Theta(N^\gamma)$ for some arbitrary $\gamma \in [0, 2]$. Let $r > 0$ be a round parameter. If* SPANNINGFOREST(G, m, r) *(Algorithm 10) does not return* FAIL*, then let the output be the input of* ORIENTATE$(\cdot)$ *(Algorithm 11), and* ORIENTATE$(\cdot)$ *can be implemented in (γ, δ) − MPC model for any constant $\delta \in (0, 1)$. Furthermore, the parallel running time is $O(R)$, where R is the total number of iterations (see Definition 4.3.21) of* SPANNINGFOREST(G, m, r).

Proof. Line 4 to line 7 can be implemented by operations described in **Multiple queries**. Notice that there is a trick here, if $f_i(u, v) = \text{null}$, we do not need to store the tuple $(\text{“}f_i\text{”}, ((u, v), \text{null}))$ in the system. The total space needed to store all the mappings f_i and all the sets F_i for $i \in \{0\} \cup [r]$

is at most $\sum_{i=0}^{r} |V_i| = O(m)$.

Line 10 and line 11 can be implemented by operations described in **Set membership** and **Multiple queries**.

We now look at the second loop, and focus on round i. Line 12 can be implemented by Lemma 4.4.9. The total space needed is at most $O(|V_i| \cdot (m/|V_i|)^{1/2} \cdot \log(m/|V_i|)) = O(m)$. The parallel running time needed is at most $O(k_i)$, where k_i is the number of iterations (see Definition 4.3.3) of MULTIPLELARGETREES(G_i, m), G_i is the intermediate graph in the procedure SPANNINGFOREST(G, m, r).

Thus, the parallel running time is $O(R)$, where R is the total number of iterations (see Definition 4.3.21) of SPANNINGFOREST(G, m, r). The total space needed is $O(m)$. □

Now, we are able to conclude the following theorem for spanning forest problem.

Theorem 4.4.12. *For any $\gamma \in [0, 2]$ and any constant $\delta \in (0, 1)$, there is a randomized $(\gamma, \delta)-$*MPC *algorithm (see Algorithm 10 and Algorithm 11) which can output the rooted spanning forest for any graph $G = (V, E)$ in $O(\min(\log D \cdot \log \frac{1}{\gamma'}, \log n))$ parallel time, where D is the diameter of G, $n = |V|, N = |V| + |E|$ and $\gamma' = (1 + \gamma) \log_n \frac{2N}{n^{1/(1+\gamma)}}$. Furthermore, the depth of the rooted spanning forest found is at most $D^{O(\log(1/\gamma'))}$. The success probability is at least* 0.98*. In addition, if the algorithm fails, then it will return* FAIL*.*

Proof. Algorithm 10 outputs all the edges in the spanning forest and all the contraction information. Algorithm 11 takes the output of Algorithm 10 as its input, and outputs a rooted spanning forest.

The implementation of Algorithm 10 and Algorithm 11 in MPC model is shown by Theorem 4.4.10 and Theorem 4.4.11 respectively. The correctness of Algorithm 10 and Algorithm 11 is proved by Corollary 4.3.17 and Theorem 4.3.19 respectively. The parallel time of Algorithm 10 and Algorithm 11 is proved by Theorem 4.3.22. By Theorem 4.3.19, the depth of that rooted spanning forest is at most $D^{O(\log(1/\gamma'))}$. □

A byproduct of our spanning forest algorithm is an estimator of the diameter of the graph.

Theorem 4.4.13. *For any $\gamma \in [0,2]$ and any constant $\delta \in (0,1)$, there is a randomized $(\gamma,\delta)-$*MPC *algorithm which can output an diameter estimator D' for any graph $G = (V,E)$ in $O(\min(\log D \cdot \log(1/\gamma'), \log n))$ parallel time such that $D \leq D' \leq D^{O(\log(1/\gamma'))}$, where D is the diameter of G, $n = |V|$, $N = |V| + |E|$ and $\gamma' = (1+\gamma)\log_n \frac{2N}{n^{1/(1+\gamma)}}$. The success probability is at least* 0.98. *In addition, if the algorithm fails, then it will return* FAIL.

Proof. By Theorem 4.4.12, we can find a rooted spanning forest. By Theorem 4.3.19, the depth of that rooted spanning forest is at most $D^{O(\log(1/\gamma'))}$. Then we can implement a doubling algorithm (e.g. Modified Lemma 4.4.6, Algorithm 6 without maintaining useless g_l) with log in depth parallel time to output the depth of that spanning forest. □

4.5 Minimum spanning forest

In this section, we discuss how to apply our connectivity/spanning forest algorithm to the Minimum Spanning Forest (MSF) and Bottleneck Spanning Forest (BSF) problem.

The input of MSF/BSF problem is an undirected graph $G = (V,E)$ together with a weight function $w : E \to \mathbb{Z}$, where E contains m edges $e_1, e_2, \cdots, e_m$ with $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$. The goal of MSF is to output a spanning forest such that the sum of weights of the edges in the forest is minimized. The goal of BSF is to output a spanning forest such that the maximum weight of the edges in the forest is minimized. D is the hop diameter of the minimum spanning forest. If there are multiple choices of the minimum spanning forest, then let D be the minimum hop diameter among all the minimum spanning forests.

For simplicity, in all of our proofs, we only discuss the case when all the edges have different weights, i.e. $w(e_1) < w(e_2) < \cdots < w(e_m)$. In this case, the minimum spanning forest is unique. It is easy to extend our algorithms to the case when there are edges with the same weight. We omit the proof for this fact.

Firstly, we show that D is an upper bound of the diameter of G' where the vertex set of G' is the vertex set of G, and the edge set of G' is $\{e_1, e_2, \cdots, e_i\}$ for some arbitrary $i \in [m]$.

Lemma 4.5.1. *Given a graph $G = (V, E)$ for $E = \{e_1, e_2, \cdots, e_m\}$ together with a weight function w which satisfies $w(e_1) < w(e_2) < \cdots < w(e_m)$, then the diameter of $G' = (V, E')$ is at most D, where D is the diameter of the minimum spanning forest of G, and E' only contains the first i edges of E, i.e. $e_1, e_2, \cdots, e_i$ for some arbitrary $i \in [m]$.*

Proof. The proof follows by Kruskal's algorithm directly. □

Our algorithms is based on the following simple but useful Lemma.

Lemma 4.5.2. *Given a graph $G = (V, E)$ for $E = \{e_1, e_2, \cdots, e_m\}$ together with a weight function w which satisfies $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$, $\forall 1 \leq i < j \leq m$, an edge e from $\{e_i, e_{i+1}, \cdots, e_j\}$ is in the minimum spanning forest of G if and only if e' from $\{e'_i, e'_{i+1}, \cdots, e'_j\}$ is in the minimum spanning forest of G', where the vertices of G' is obtained by contracting all the edges $e_1, e_2, \cdots, e_{i-1}$ of G, and $e', e'_i, e'_{i+1}, \cdots, e'_j$ are the edges (or vertices) in G' which corresponds to the edges $e, e_i, e_{i+1}, \cdots, e_j$ before contraction.*

Proof. The proof follows by Kruskal's algorithm directly. □

A natural way to apply Lemma 4.5.2 to parallel minimum spanning forest algorithm is that we can divide the edges into several groups, and recursively solve the minimum spanning forest for each group of edges. More precisely, suppose we have total space $\Theta(km)$, we can divide E into k groups $E_1, E_2, \cdots, E_k$, where $E_i = \{e_{(i-1)\cdot m/k+1}, e_{(i-1)\cdot m/k+2}, \cdots, e_{i\cdot m/k}\}$. We can compute graph $G_1, G_2, \cdots, G_k$ where the vertices of G_i is obtained by contracting all the edges from e_1 to $e_{(i-1)\cdot m/k}$, the edges of G_i are corresponding to the edges in E_i. Then by Lemma 4.5.2, we can obtain the whole minimum spanning forest by solving these k size $O(m/k)$ minimum spanning forest problems. For each sub-problem, we can assign it $\Theta(m)$ working space, thus each sub-problem still has $\Theta(k)$ factor more total space. Therefore, we can recursively apply the above argument.

Theorem 4.5.3. *For any $\gamma \in [0, 2]$ and any constant $\delta \in (0, 1)$, there is a randomized (γ, δ) –* MPC *algorithm which can output a minimum spanning forest for any weighted graph $G = (V, E)$*

with weights $w : E \to \mathbb{Z}$ *in* $O(\min(\log D \cdot \log(1/\gamma'), \log n) \cdot 1/\gamma')$ *parallel time, where* $n = |V|$, $\forall e \in E, |w(e)| \leq \text{poly}(n)$, D *is the diameter of a minimum spanning forest of* G, *and* $\gamma' = \gamma/2 + \Theta(1/\log n)$. *The success probability is at least* 0.98. *In addition, if the algorithm fails, then it will return* FAIL.

Proof. Let $n = |V|, m = |E|$. Let $E = \{e_1, \cdots, e_m\}$ with $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$. The total space in the system is $\Theta(m^{1+\gamma})$. Let $k = \Theta(m^{\gamma/2})$. By our previous discussion, we can divide E into k groups $E_1, E_2, \cdots, E_k$, where $E_i = \{e_{(i-1)\cdot m/k+1}, e_{(i-1)\cdot m/k+2}, \cdots, e_{i\cdot m/k}\}$. By Lemma 4.5.1 and Theorem 4.4.4, we can use $O(\min(\log D \cdot \log(1/\gamma'), \log n))$ parallel time and $\Theta(km^{1+\gamma/2})$ total space to compute graph $G_1, G_2, \cdots, G_k$ where the vertices of G_i is obtained by contracting all the edges from e_1 to $e_{(i-1)\cdot m/k}$, the edges of G_i are corresponding to the edges in E_i after contraction.

By Lemma 4.5.2, it suffices to recursively solve the minimum spanning forest problem for each group G_i. Since each time, we split the edges into k groups, the recursion will have at most $O(1/\gamma')$ levels. At the end of the recursion, we are able to determine for every edge e whether e is in the minimum spanning forest.

Now let us consider the success probability. Although Theorem 4.4.4 is a randomized algorithm, the parallel time is always bounded by $\min(\log D \cdot \log(1/\gamma'), \log n)$. If we repeat the algorithm until it succeeds, the expectation of number of trials is a constant. Furthermore, for each level of the recursion, we can regard the graphs in all the tasks composed one large graph. Thus, in real implementation, in each level of the recursion, we will only invoke one connectivity procedure. Thus in expectation, the total parallel time is $O(\min(\log D \cdot \log(1/\gamma'), \log n) \cdot 1/\gamma')$. By applying Markov's inequality, we complete the proof. □

In the following theorem, we show that Lemma 4.5.2 can also be applied in approximate minimum spanning forest problem.

Theorem 4.5.4. *For any* $\gamma \in [\beta, 2]$ *and any constant* $\delta \in (0, 1)$, *there is a randomized* (γ, δ) – MPC *algorithm which can output a* $(1 + \epsilon)$ *approximate minimum spanning forest for any weighted graph* $G = (V, E)$ *with weights* $w : E \to \mathbb{Z}_{\geq 0}$ *in* $O(\min(\log D \cdot \log(1/\gamma'), \log n))$ *parallel time,*

where $n = |V|$, $N = |V| + |E|$, $\beta = \Theta(\log(\epsilon^{-1}\log n)/\log n)$, $\forall e \in E, |w(e)| \leq \text{poly}(n)$, D *is the diameter of a minimum spanning forest of* G, *and* $\gamma' = (1+\gamma-\beta)\log_n \frac{2N}{n^{1/(1+\gamma-\beta)}}$. *The success probability is at least* 0.98. *In addition, if the algorithm fails, then it will return* FAIL.

Proof. For each edge $e \in E$, we can round $w(e)$ to $w'(e)$ such that $w'(e) = 0$ when $w(e) = 0$, and $w'(e) = (1+\epsilon)^i$ when $w(e) \neq 0$, and i is the smallest integer such that $w(e) \leq (1+\epsilon)^i$.

Since $|w(e)| \leq \text{poly}(n)$ for all $e \in E$, there are only $k = O(\log(n)/\epsilon)$ different values of $w'(e)$. We can divide E into k groups, where the i^{th} group E_i contains all edges with the i^{th} largest weight in w'. By Lemma 4.5.1 and Theorem 4.4.4, we can use $O(\min(\log D \cdot \log(1/\gamma'), \log n))$ parallel time and $\Theta(kN^{1+\gamma-\beta}) = \Theta(N^{1+\gamma})$ total space to compute graph $G_1, G_2, \cdots, G_k$ where the vertices of G_i is obtained by contracting all the edges from E_1 to E_{i-1}, the edges of G_i are corresponding to the edges in E_i after contraction.

Then, for each G_i, since all the edges have the same w' weight, any spanning forest of G_i is a minimum spanning forest of G_i. By Theorem 4.4.12, we can use $O(\min(\log D \cdot \log(1/\gamma'), \log n))$ parallel time and $\Theta(kN^{1+\gamma-\beta}) = \Theta(N^{1+\gamma})$ total space to compute the spanning forest for each graph $G_1, G_2, \cdot, G_k$. By Lemma 4.5.2, the union of all the minimum spanning forest with respect to w' must be the minimum spanning forest of G with respect to w'. Since all the weights w are nonnegative integers, w' is a $(1+\epsilon)$ approximation to w. Therefore, our output minimum spanning forest with respect to w' is a $(1+\epsilon)$ approximation to the minimum spanning forest with respect to w.

For the success probability, we can apply the similar argument made in the proof of Theorem 4.5.3 to prove that the success probability is at least 0.98. □

In the following, we show that if we only need to find the largest edge in the minimum spanning tree, then we are able to get a better parallel time. It is an another application of our

Theorem 4.5.5. *For any* $\gamma \in [0,2]$ *and any constant* $\delta \in (0,1)$, *there is a randomized* $(\gamma, \delta) - \text{MPC}$ *algorithm which can output a bottleneck spanning forest for any weighted graph* $G = (V, E)$ *with weights* $w : E \to \mathbb{Z}$ *in* $O(\min(\log D \cdot \log(1/\gamma'), \log n) \cdot \log(1/\gamma'))$ *parallel time, where* $n = |V|$,

$\forall e \in E, |w(e)| \leq \text{poly}(n)$, *$D$ is the diameter of a minimum spanning forest of G, and* $\gamma' = \gamma/2 + \Theta(1/\log n)$. *The success probability is at least* 0.98. *In addition, if the algorithm fails, then it will return* FAIL.

Proof. Let $n = |V|, m = |E|$. Let $E = \{e_1, \cdots, e_m\}$ with $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$. The total space in the system is $\Theta(m^{1+\gamma})$. Let $k = \Theta(m^{\gamma/2})$. By our previous discussion, we can divide E into k groups $E_1, E_2, \cdots, E_k$, where $E_i = \{e_{(i-1)\cdot m/k+1}, e_{(i-1)\cdot m/k+2}, \cdots, e_{i\cdot m/k}\}$. By Lemma 4.5.1 and Theorem 4.4.4, we can use $O(\min(\log D \cdot \log(1/\gamma'), \log n))$ parallel time and $\Theta(km^{1+\gamma/2})$ total space to compute graph $G_1, G_2, \cdots, G_k$ where the vertices of G_i is obtained by contracting all the edges from e_1 to $e_{(i-1)\cdot m/k}$, the edges of G_i are corresponding to the edges in E_i after contraction.

By Lemma 4.5.2, the edge with largest weight must be in the group E_i for some i with $G_{i+1} = G_{i+2}$. Thus, we reduce the problem size to m/k. By double-exponential speed problem size reduction technique described in Chapter 3, we can finish the recursion in $O(\log(1/\gamma'))$ phases.

Suppose the bottleneck is e_i, then by Theorem 4.4.12, we can find a spanning forest by only using edges from $\{e_1, \cdots, e_i\}$ in $O(\min(\log D \cdot \log(1/\gamma'), \log n))$ parallel time and in $\Theta(m^{1+\gamma/2})$ total space. Thus, the resulting spanning forest is a bottleneck spanning forest.

For the success probability, we can apply the similar argument made in the proof of Theorem 4.5.3 to prove that the success probability is at least 0.98. □

4.6 Connectivity and spanning forest in PRAM

In this section, we show how to implement the truncated broadcasting, the connectivity algorithm and the spanning forest algorithm in a more careful way such that they can be implemented in PRAM using linear processors and small parallel time as the same as the MPC implementation.

4.6.1 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 $v = v.p$. The root of a tree here denotes a leader vertex mentioned in previous sections. 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, 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*.

4.6.2 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 [77, 78, 79], not denser by adding edges.)

We treat each undirected 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. Concurrent direct links maintain monotonicity.

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. [80] introduced shortcutting in their connected components algorithm; it is closely related to the *compress* step in tree contraction [81] and to *path splitting* in disjoint-set union [82].

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 [83] 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. This operation corresponds to truncated broadcasting (Algorithm 1). To use a small depth and work, we need to implement the procedure in a more careful way. The key idea for implementing this expansion procedure 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.

4.6.3 Connectivity in ARBITRARY CRCW PRAM

In this section we present our connected components algorithm. For simplicity in presentation, we first consider the COMBINING CRCW PRAM [84], 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 [85] but adapted in our frame-

work, which is called Vanilla algorithm. We implement the algorithm on a COMBINING CRCW PRAM and then generalize it to run on an ARBITRARY CRCW PRAM.

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

Lemma 4.6.2. *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 4.6.3. *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 4.6.2, 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 4.6.3, the following corollary is immediate by linearity of expectation and Markov's inequality:

Corollary 4.6.4. *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 4.6.2, Corollary 4.6.4, and monotonicity, we have that Vanilla algorithm outputs the connected components in $O(\log n)$ time with high probability.

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 (altered) 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 4.6.2 also holds for this algorithm.

The details of the EXPAND and VOTE will be presented later. We call an iteration of the repeat loop after the PREPARE a *phase*. By Lemma 4.6.2, 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 4.6.5. *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 4.6.4, 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 4.6.6. *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 we will show how to remove Assumption 4.6.6 to implement our algorithms on an ARBITRARY CRCW PRAM.

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.

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 4.6.6), 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 $H(u)$. 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 4.6.7. *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 4.6.8. *The* EXPAND *takes $O(\log d)$ time, where d is the (hop) diameter of the original graph.*

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 4.6.7. □

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 4.6.9. *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 4.6.10. *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 4.6.7, 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 4.6.9 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 4.6.7, 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 $b := \delta^{1/18}$, 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 4.6.11. *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

$$\frac{n' - 1}{m/b^{12}} \leq \frac{n'}{n'b^6} = \frac{1}{b^6}, \tag{4.1}$$

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 4.6.11, 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 4.6.12. $|B(u, r-1)| \leq |H_i(u)|$.

Proof. Let w be a vertex in $B(u, r-1)$. If $\text{dist}(w, u) \leq 2^{i-1}$ then $w \in H_{i-1}(u)$ by Lemma 4.6.7 and Definition 4.6.9, and position $h_V(w)$ in $H_i(u)$ remains occupied in round i. Suppose $\text{dist}(w, u) > 2^{i-1}$. Let x be a vertex on the shortest path from u to w and x satisfies $\text{dist}(x, w) = 2^{i-1}$. We obtain $B(x, 2^{i-1}) \subseteq B(u, r-1)$ since any $y \in B(x, 2^{i-1})$ has $\text{dist}(y, u) \leq \text{dist}(y, x) + \text{dist}(x, u) \leq 2^{i-1} + r - 1 - 2^{i-1}$, 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 4.6.12 holds. □

Lemma 4.6.13. *After the* EXPAND*, for any dormant vertex u, $|H(u)| < b$ with probability at most b^{-1}.*

Proof. By Lemma 4.6.11, $|B(u,r)| > b^2$ 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 4.6.10, 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 4.6.12. 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 4.6.9, 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 4.6.10. It follows that $w_{i-2} \in H_{i-1}(u)$ by Lemma 4.6.7 and Definition 4.6.9. We also need the following claim, which follows immediately from $\mathrm{dist}(w_j, u) \leq r - 2^j$ and Definition 4.6.9:

Claim 4.6.14. *$H_j(w_j) = B(w_j, 2^j)$ 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 $\mathrm{dist}(w_{j-1}, w_j) = 2^j$ and Claim 4.6.14, 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 4.6.13. □

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 4.6.7, $H(u)$ 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 4.6.13, $|H(u)| \geq b$ 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

$$\frac{n' \cdot b^{-1/2}}{n' \cdot b^{-1/4}} \leq b^{-1/4}. \tag{4.2}$$

Removing the assumption

In this section, we remove Assumption 4.6.6 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 4.6.11). Using $\widetilde{n}$ to rewrite Inequality (4.1), 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 4.6.11 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 (4.2) 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 4.6.15. *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 4.6.15, and a union bound, the following lemma is immediate:

Lemma 4.6.16. *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 $i \geq 1$.*

Running time

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

Lemma 4.6.17. *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 4.6.2 and monotonicity, the algorithm terminates and outputs the correct connected components in this phase since no parent changes. □

Theorem 4.6.18 (Connectivity)**.** *There is an ARBITRARY CRCW PRAM algorithm using $O(m)$ processors that computes the connected components of any given graph. With probability $1 - 1/poly(((m + n) \log n)/n)$, it runs in $O(\log(d + 1) \log \log_{2(m+n)/n} n)$ time.*

Proof. We set $c = 100$ in the PREPARE. If $m/n > \log^c n$, then $\widetilde{n} = n$ in the first phase by Lemma 4.6.5 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 4.6.16, $\widetilde{n} \geq n'$ 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 4.6.17, the number of phases is $O(\log \log_{m/n} n)$. By Lemma 4.6.8, 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 4.6.5 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 $O(\log \log n) + O(\log d \cdot \log \log_{m/n_1} n) = O(\log d \log \log_{m/n} n)$. □

4.6.4 Spanning forest in ARBITRARY CRCW PRAM

Many existing connected components algorithm can be directly transformed into a spanning forest algorithm. For example in Reif's algorithm [85], 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 shown in our MPC spanning forest uses several subroutines including computing the distances by truncated broadcasting, 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 [86].

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 (d here denotes the (hop) diameter of the original graph) 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.

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 4.6.2 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 4.6.19. *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 4.6.20. *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 4.6.2, 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 4.6.21. *For any positive integer j, F_j is a forest. And in each tree of F_j, there is exactly one root.*[1]

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 4.6.20, 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 $u.p = u$. □

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.

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

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 before, 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.

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 $Q'(u)$.

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 $u.\widehat{e}.f \coloneqq 1$.

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*.

Lemma 4.6.22. *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 4.6.7. 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 4.6.23. *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 4.6.22, $Q(u) = B(u, u.\alpha)$ 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 4.6.24. *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 4.6.22, $Q(u) = B(u, 0) = \{u\}$. 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 4.6.23, there is a leader v such that $\text{dist}(u, v) = u.\beta$. Let $\{u, w\}$ be a graph edge with $\text{dist}(w, v) = u.\beta - 1$, 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 4.6.22, we have that $w.\alpha = u.\beta - 2$. Let $\{x, v\}$ be a graph edge such that $\text{dist}(w, x) = u.\beta - 2$, 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 4.6.25. *For any vertex u, if $u.\beta > 0$, then u is finished in the next phase.*

Lemma 4.6.26. *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 4.6.23 and the fact that an ALTER and adding edges never increase the diameter, the height of any tree is at most d. □

As mentioned before, by the above lemma and Lemma 4.6.8, the following is immediate:

Corollary 4.6.27. *Each phase of the algorithm takes $O(\log d)$ time.*

Similar to Definition 4.6.19 and Lemma 4.6.21, we can show the following, which guarantees the correctness of Spanning Forest algorithm:

Lemma 4.6.28. *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.*

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 4.6.9 with respect to u.

Lemma 4.6.29. *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 4.6.22, we have that $u.\alpha = \mathrm{dist}(u, v) - 1$, and $Q(u) = B(u, \mathrm{dist}(u, v) - 1)$ which contains a leader-neighbor. By Step (4b) in the TREE-LINK, $u.\beta = \mathrm{dist}(u, v) > 0$. By Corollary 4.6.25, u is finished in the next phase. □

Using the above lemma and Corollary 4.6.27, the remaining analysis is almost identical to that in Connected Components algorithm.

Theorem 4.6.30 (Spanning forest)**.** *There is an ARBITRARY CRCW PRAM algorithm using $O(m)$ processors that computes the spanning forest of any given graph. With probability $1-1/poly((2(m+n)\log n)/n)$, it runs in $O(\log(d+1)\log\log_{2(m+n)/n} n)$ time.*

Proof. By Lemma 4.6.11, 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 *Removing the assumption* and *Running time* in Section 4.6.3, 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 4.6.27). □

4.6.5 Connectivity in COLLISION CRCW PRAM

In this section, we show that our connectivity algorithm can be further implemented in the COLLISION CRCW PRAM model which is much weaker model than ARBITRARY CRCW PRAM. In the COLLISION CRCW PRAM model, if multiple processors are trying to write the same cell at the same time, then the cell can be written an arbitrary value and there is a flag indicating that the cell is written by multiple processors.

Simulation of concurrent writing in COLLISION CRCW PRAM

In this section, we show a randomized COLLISION CRCW subroutine which can simulate the simultaneous writing of ARBITRARY CRCW PRAM.

Lemma 4.6.31 (Balls and bins)**.** *If we throw k balls independently at random into k' bins for $k \leq k'$. With probability at least $1 - k/k'$, there exists at least one bin which contains exactly one*

ball.

Proof. Consider the first ball, the probability that it is in the same bin as the i-th ball is $1/k'$. By union bound, the probability that the first ball collides an another ball is at most k/k'. Thus, with probability at least $1 - k/k'$, the bin containing the first ball only contains the first ball. □

Lemma 4.6.32. *Let $n \geq 1$. Let $c > 0$ be an arbitrary constant. Let $k \in [c'' \log n, n^c]$ for a sufficiently large constant $c'' > 0$ only depending on c. Consider a shared memory cell C in the ARBITRARY CRCW PRAM model. Consider m processors which may potentially write C at the same time, where $m \leq n^c$. Then, each concurrent write on C in the ARBITRARY CRCW PRAM can be simulated in the COLLISION CRCW PRAM model in $O(1)$ parallel time with probability at least $1 - 1/k$ if we can use additional $\Theta(k^6)$ processors associated with the cell C. If the simulation fails, the process will output FAIL.*

Proof. Let $p_1, p_2, \cdots, p_m$ be the processors which may potentially write the target shared memory cell C in the ARBITRARY CRCW PRAM model.

Without loss of generality, we can suppose that at least one of $p_1, \cdots, p_m$ is going to write C due to the following reason. For each processor p_i, if it tries to write C, then it directly writes C. If there is no collision, it is done. Otherwise, we know that at least 2 processors are trying to write C. Then we do the following procedure to simulate concurrent write.

We create $L = \lceil c \log(n) \rceil + 1$ groups of bins, where the groups are indexed from $\{0, 1, \cdots, L\}$. Each group contains k^2 bins. Here each bin in each group corresponds to a shared memory cell in the COLLISION CRCW PRAM model.

In the following, we describe the simulation process.

1. For each processor p_i, if p_i is going to write C, let p_i write a random bin of the 0-th group and also let p_i choose another group $j \in [L]$ where j is chosen with probability $1/2^j$,[2] and write a random bin of the j-th group.

[2] With probability $1/2^L$, the processor may not choose any j.

2. Look at all bins over all groups. Find the bin which is successfully written (written by exactly one processor of $p_1, p_2, \cdots, p_m$) and has the smallest index (the index of a bin is first ranked by the index of its group and then its index in the group). Write the value from that bin to C. If such bin does not exist, output FAIL.

Claim 4.6.33. *The above process can be simulated in the COLLISION CRCW PRAM model in $O(1)$ parallel time using additional $O(L \cdot k^2) = O(k^3)$ additional shared memory cells and $O((L \cdot k^2)^2) = O(k^6)$ additional processors.*

Proof. It is clear that step 1 only needs $O(L \cdot k^2)$ additional total space to represent bins, and it does not need any additional processors. The parallel time of step 1 is $O(1)$. For step 2, we assign each pair of bins an additional processor. If bin w is successfully written (written by exactly one processor from $p_1, p_2, \cdots, p_m$) and its index is smaller than bin z, mark a *flag* for z. Then we assign each bin a processor. If bin w is not marked a flag (if there is a collision for the flag, then it is known that some processors mark a flag for w) and bin w is successfully written, then write the value of bin w to C. Consider the bin x which is successfully written and has the smallest index. We know that every bin with index larger than x will be marked a flag and x cannot be marked a flag. If C is not written at the end, we can output FAIL. Thus, step 2 can be simulated. In addition, step 2 uses additional $O((L \cdot k^2)^2)$ processors and can be done in $O(1)$ parallel time. □

The only thing remaining is to prove the success probability. It suffices to show that with probability at least $1 - 1/k$, there is at least one bin such that it is written by exact one processor of $p_1, \cdots, p_m$.

Let q be the number of processors of $p_1, p_2, \cdots, p_m$ that is trying to write C. Consider the first case: $q \leq k$. Since the number of bins in 0-th group is k^2, according to Lemma 4.6.31, the probability that at least one bin that is written by exact one processor of $p_1, \cdots, p_m$ is at least $1 - k/k^2 = 1 - 1/k$. Consider the second case: $q > k$. We can find $i \in [L]$ such that $q/2^i \in [k/8, k/4]$. Since $k \geq c'' \cdot \log n$, according to Chernoff bound, with probability at least $1 - 1/(2n^c) \geq 1 - 1/(2k)$, the number of processors of $p_1, p_2, \cdots, p_m$ that is writing some bin of the i-th group

is in $[k/16, k/2]$. According to Lemma 4.6.31, conditioning on that the number of processors of $p_1, p_2, \cdots, p_m$ that is writing some bin of the i-th group is in $[k/16, k/2]$, the probability that at least one bin that is written by exact one processor is at least $1 - 1/(2k)$. By taking union bound, with probability at least $1 - 1/k$, there is at least one bin such that it is written by exact one processor of $p_1, p_2, \cdots, p_m$. □

Lemma 4.6.34. *Consider a shared memory cell C in the ARBITRARY CRCW PRAM model. Consider k processors which may potentially write C at the same time. Then each concurrent write on C in the ARBITRARY CRCW PRAM can be simulated in the COLLISION CRCW PRAM model in $O(1)$ parallel time if we can use additional $\Theta(k^2)$ processors.*

Proof. Let $p_1, p_2, \cdots, p_k$ be the processors which may potentially write the target shared memory cell C. For each pair $i, j \in [k]$, we assign a processor. If p_i tries to write C, and $i < j$, mark j a flag. Finally, if p_i wants to write C and i is not marked a flag, let p_i write C. The parallel time is $O(1)$ and the number of additional processors used is $\Theta(k^2)$. □

Vanilla algorithm in COLLISION CRCW PRAM

Let us first look back to the vanilla algorithm described in Section 4.6.3. The main challenge of implementing vanilla algorithm in the COLLISION CRCW PRAM model is that if there are multiple arcs (v, w) such that v is not a leader and w is a leader, i.e., $v.l = 0, w.l = 1$ then there are multiple processors which may update $v.p$ which may cause writing conflict. Let $L = \lceil \log m \rceil$. For each vertex v, we add additional L variables $v.p_1, \cdots, v.p_L$. These variables are used to update $v.p$ We say that a variable/shared memory cell is successfully written if it is written by exact one processor. We modify the vanilla algorithm as the following:

Vanilla-COLLISION 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:

1. For each graph arc (v, w):

   (a) If $v.l = 0$ and $w.l = 1$ then sample $j \in [L]$ with probability $1/2^j$ (with probability $1/2^L$, none of j is sampled). If j is sampled, write w to $v.p_j$.

   (b) If $v.p_j$ is successfully written:

      i. If none of $v.p_{j+1}, v.p_{j+2}, \cdots, v.p_{\min(j+10,L)}$ was successfully written by other processors in step 1a, write w to $v.p$.

      ii. Clear the writing value of $v.p_j$.

2. For each vertex v, if writing conflict happens for $v.p$, recover $v.p$ as its previous value.

SHORTCUT: for each vertex u: update $u.p$ to $u.p.p$.

ALTER: for each edge $e = \{v, w\}$: replace it by $\{v.p, w.p\}$.

Since the writing value is always cleared at the end of LINK operation when $v.p_j$ is successfully written, at any time outside the LINK operation, every $v.p_j$ is either marked writing conflict or not written any value. Therefore, in step 1(b)i of LINK, if $v.p_{j+x}$ has a written value, it must be written by another processor in step 1a. To implement step 2 of LINK, we just need to make a copy of $v.p$ for each vertex v before LINK.

It is easy to see that Vanilla algorithm uses $O(m + n)$ processors since we only need to assign one processor for each vertex and one processor for each arc. We call an iteration of the repeat loop in the algorithm a *phase*. Clearly each phase takes $O(1)$ parallel time.

We follow Definition 4.6.1 to define ongoing vertices. Lemma 4.6.2 still holds for Vanilla-COLLISION algorithm. Next, we prove an analog of Lemma 4.6.3.

Lemma 4.6.35. *Given a vertex u, after k phases of Vanilla-COLLISION algorithm, u is ongoing with probability at most* $(31/32)^k$

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 $(31/32)^{k-1}$. Furthermore, by Lemma 4.6.2, there must be an edge $\{u, v\}$ such that v is ongoing. Thus, with probability at least $1/4$, there is at least one arc (u, w) such that $u.l = 0$ and $w.l = 1$. We condition on this event. Suppose the number of arcs (u, w) with $u.l = 0, w.l = 1$ is q. let $j' \in [L]$ satisfy $q \in [2^{j'-1}, 2^{j'}]$. Then the probability that $u.p_{j'}$ is written by exact one processor and none of $u.p_j$ for $j > j'$ is written is

$$\begin{aligned} & q \cdot 1/2^{j'} \cdot (1 - 1/2^{j'+1})^{q-1} \\ \geq & \frac{1}{2} \cdot (1 - 1/2^{j'+1})^{2^{j'+1}} \\ \geq & \frac{1}{4} \end{aligned}$$

Now consider $j \in [L]$ and $q \geq 100 \cdot 2^j$. The expected number of processors which write $u.p_j$ is $q/2^j \geq 100$. The variance of number of processors that write $u.p_j$ is at most $q/2^j$. By Chebyshev's inequality, the probability that the number of processors which write $u.p_j$ is at least 2 is at least $2^{j+2}/q$. By taking union bound, with probability at least $1 - 1/8$, $\forall j \in [L]$ with $q \geq 100 \cdot 2^j$, writing conflict must happen for $u.p_j$.

Thus, with probability at least $1/8$, the following things happens:

1. $u.p_{j'}$ is written by exact one processor and none of $u.p_j$ for $j > j'$ is written.
2. $\forall j \in [L]$ with $q \geq 100 \cdot 2^j$, writing conflict happens for $u.p_j$.

According to step 1(b)i, with probability at least $1/8$, $u.p$ is written by exact one processor — the processor which successfully writes $u.p_{j'}$. Therefore, with probability at least $1/32$, a ongoing vertex u after phase $k - 1$ is finished after phase k. It follows that the probability that after phase k, a vertex u is still ongoing is at most $(31/32)^k$. □

By Lemma 4.6.35, the following corollary is immediate by linearity of expectation and Markov's inequality:

Corollary 4.6.36. *After k phases of Vanilla-COLLISION algorithm, the number of ongoing vertices is at most $(63/64)^k n$ with probability at least $1 - (62/63)^k$.*

Thus, we know that Vanilla-COLLISION algorithm outputs the connected components in $O(\log n)$ time with high probability.

Algorithmic framework

The algorithmic framework of our COLLISION CRCW PRAM connectivity algorithm is almost the same as the algorithmic frame work shown in Section 4.6.3.

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

PREPARE-COL: if $m/n \leq \log^c n$ for given constant c then run $c \log_{63/62} \log n$ phases of Vanilla-COLLISION algorithm.

EXPAND-COL: 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-COL: 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-COLLISION algorithm. The VOTE is the same as the VOTE of the Connected Components algorithm described in Section 4.6.3. We will see later that the outcome of LINK-COL is the same as the LINK of the Connected Components algorithm described in Section 4.6.3. Therefore, Lemma 4.6.2 also holds for this algorithm.

The details of the EXPAND-COL and LINK-COL will be presented later. We call an iteration of the repeat loop after the PREPARE-COL a *phase*. By Lemma 4.6.2, 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, 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-COL before the main loop to obtain a large enough δ with good probability:

Lemma 4.6.37. *After the* PREPARE-COL*, 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 proof is similar to the proof of Lemma 4.6.5. The first part is trivial since the PREPARE-COL does nothing. By Corollary 4.6.36, after $c \log_{63/62} \log n$ phases, there are at most $n/\log^c n$ ongoing vertices with probability at least $1 - 1/\log^c n$, and the lemma follows immediately from $m \geq n$. □

We will be cocusing on EXPAND-COL, VOTE, and LINK-COL. 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 graph, call the current edge just the edge, and call an ongoing vertex just a vertex.

In the following algorithms and analyses, we will first assume Assumption 4.6.6 for simplicity in the analyses. Later we will show how to remove the assumption.

The expansion

In this section, we present the method EXPAND-COL and show that all vertices have a large enough neighbor set after the EXPAND-COL with good probability. Two concepts **blocks** and **hashing** are very similar to the concepts that are described for EXPAND in Section 4.6.3.

Blocks. We shall use a pool of $\Theta(m)$ processors to do the EXPAND-COL. We divide these into $m/\delta^{13/14}$ indexed *blocks*, where each block contains $\Theta(\delta^{13/14})$ indexed processors. Since n' and

δ are known at the beginning of each phase (cf. Assumption 4.6.6), if a vertex is assigned to a block, then it is associated with $\delta^{13/14}$ (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 $\delta^{1/14}$, because we will see later that we need $\delta^{6/7}$ processors for each cell in the table to do an expansion step. 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 $H(u)$. If u does not own a block, we think that $H(u) = \emptyset$.

Collision and writing conflict. In our presentation, a writing conflict corresponds to the event that multiple processors want to write the same cell. A collision corresponds to the event that some vertices u, v have $h_V(u) = h_V(v)$.

Let $b = \delta^{1/84} = (m/n')^{1/84}$, i.e., $m = n' \cdot b^{84}$. Thus, we know that $h_B : [n] \to [m/b^{78}]$ and $h_V : [n] \to [b^6]$. Each block contains $\Theta(b^{78})$ indexed processors. We present the method EXPAND-COL as follows. The major differences between EXPAND-COL and EXPAND from Section 4.6.3 is marked in bold.

EXPAND-COL:

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 each vertex v that does not own a block as *dormant* and set $H(v) = \{v\}$.
3. **For each live vertex v, assign $\Theta(b^{72})$ indexed processors for each cell in the table $H(v)$.**
4. For each graph arc (v, w): if v is live before Step (4) then **use the procedure described**

by Lemma 4.6.32 with $k = b^{12}$ to write v into the $h_V(v)$-th cell of $H(v)$ and write w into the $h_V(w)$-th cell of $H(v)$.

5. **For each live vertex v before Step (5), if some cell of $H(v)$ was going to be written some value in Step (4) but failed (see FAIL in Lemma 4.6.32), mark v as dormant and set $H(v) = \{v\}$.**

6. For each live vertex v before Step (6), if there is a vertex $u \in H(v)$ which is marked as dormant before Step (6), mark v as dormant.

7. For each hashing done in Step (4): if it causes a *collision* (a cell was trying to be written by different values) in $H(v)$ then mark v as *dormant*.

8. Repeat the following until there is neither live vertex nor hash table getting a new entry:

   (a) For each live vertex u before Step (8a) in this iteration: for each v in $H(u)$: if v is dormant before Step (8a) in this iteration then mark u as *dormant*, for each w in $H(v)$: write w to the $h_V(w)$-th cell of $H(u)$ **by using the procedure described in Lemma 4.6.34 with $k = \Theta((b^6)^2)$.**

   (b) For each hashing done in Step (8a): if it causes a collision in $H(u)$ then mark u as *dormant*.

Notice that since if a vertex v owns a block, there are $\Theta(b^{78})$ indexed processors in the block. The number of cells in table $H(v)$ is b^6. According to Lemma 4.6.32, simulating the concurrent write of all graph arc processors on all cells of $H(v)$ with $k = b^{12}$ needs $O(b^6 \cdot k^6) = O(b^{78})$ processors. Thus, the number of processors in a block is larger than the number of processors required in Step (4). Consider Step (8a). The number of processors which may potentially write a cell in $H(u)$ is at most $(b^6)^2 = O(b^{12})$ According to Lemma 4.6.34, simulating the concurrent write of $(b^6)^2$ processors on all cells of $H(u)$ needs $O(b^6 \cdot (b^{12})^2) = O(b^{30})$ processors. Thus, the number

of processors in a block is larger than the number of processors required in Step (8a).

According to Lemma 4.6.32 and Lemma 4.6.34, the first seven steps and each iteration of Step (8) in the EXPAND-COL take $O(1)$ time. We call an iteration of the repeat loop in Step (8) a *round*. We say a statement holds before round 0 if it is true before Step (6), it holds in round 0 if it is true after Step (7) and before Step (8), and it holds in round i ($i > 0$) if it is true just after i iterations of the repeat loop in Step (8).

Additional notations. We follow the same notation $\text{dist}(u, v), B(u, \alpha), H_j(u)$ used in Section 4.6.3. The definition of *fully dormant* is slightly different from that was defined in Section 4.6.3. Consider a vertex u that is dormant after the EXPAND-COL. We call u *fully dormant* if u is dormant before round 0, i.e., either u does not own a block or **the concurrent write on some cell in $H(u)$ in Step (4) fails (see Lemma 4.6.32 for the definition of FAIL)**. 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-COL, let $i \geq 0$ be the first round that its hash table is the same as the table just before round i. It is easy to verify that the proof of Lemma 4.6.7 still holds.

Lemma 4.6.38 (Restatement of Lemma 4.6.7)**.** *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 $j \in [0, i-1]$, $H_j(u) = B(u, 2^j)$.*

It is easy to verify that the proof of Lemma 4.6.8 still holds.

Lemma 4.6.39 (Similar statement of Lemma 4.6.8)**.** *The* EXPAND-COL *takes $O(\log d)$ time, where d is the (hop) diameter of the input graph.*

We want to show that the table of u in round i contains enough vertices, 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.

Let r be the same as defined in Definition 4.6.9, i.e., for any vertex u that is dormant after the EXPAND-COL, let r be the minimal integer such that there is no collision nor fully dormant vertex in $B(u, r-1)$.

It is easy to verify that the proof of Lemma 4.6.10 still holds.

Lemma 4.6.40 (Restatement of Lemma 4.6.10)**.** *If u is fully dormant then $r = 0$. If u is half dormant then $2^{i-1} < r \leq 2^i$.*

Lemma 4.6.41. *A vertex u is fully dormant with probability at most $2/b^6$.*

Proof. The probability that $\exists v \neq u, h_B(v) = h_B(u)$ is at most

$$\frac{n'-1}{m/b^{78}} \leq \frac{n'}{n'b^6} = \frac{1}{b^6}. \tag{4.3}$$

Thus with probability at least $1-1/b^6$, u owns a block. Consider Step (4). According to Lemma 4.6.32, the probability that the concurrent write is failed to simulate for a cell in $H(u)$ is at most $1/b^{12}$. Since there are b^6 cells in the table $H(u)$, by union bound, the probability that none of concurrent writes for any cell of $H(u)$ is failed is at least $1 - 1/b^6$. Thus, the overall probability that vertex u is fully dormant is at most $2/b^6$. □

By plug Lemma 4.6.41 into the proof of Lemma 4.6.11 to bound the probability that a vertex is fully dormant, we can verify that the proof of Lemma 4.6.11 still holds.

Lemma 4.6.42 (Restatement of Lemma 4.6.11)**.** *For any vertex u that is dormant after the* EXPAND-COL*, $|B(u, r)| \leq b^2$ with probability at most b^{-2}.*

It is easy to verify that the proof of Lemma 4.6.12 still holds.

Lemma 4.6.43 (Restatement of Lemma 4.6.12)**.** $|B(u, r-1)| \leq |H_i(u)|$.

Thus, we are able to verify that the proof of the core lemma Lemma 4.6.13 still holds.

Lemma 4.6.44 (Restatement of Lemma 4.6.13)**.** *After the* EXPAND-COL*, for any dormant vertex u, $|H(u)| < b$ with probability at most b^{-1}.*

The voting and the link

In this section, we present the method VOTE, the method LINK-COL, 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-COL 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 $1 - b^{-2/3}$.

Notice that if u is live, then u owns a block which contains $\Theta(b^{78})$ indexed processors. To implement Step (1), for each pair of cells of table $H(u)$, we need a processor. Thus, we only need b^{12} processors. Thus, the processors in the block owned by u is enough.

LINK-COL: For each u which is not fully dormant, for each v in $H(u)$, if $v.l = 1$, use Lemma 4.6.34 with $k = b^6$ to write $u.p := v$.

Notice that if u is not fully dormant, then u owns a block containing $\Theta(b^{78})$ indexed processors. Notice that the table $H(u)$ has size at most b^6. According to Lemma 4.6.34, writing $u.p$ only requires $O(b^{12})$ processors. Thus, the processors in the block owned by u is enough.

Now let us analyze the number of ongoing vertices remained for the next phase.

There are two cases depending on whether u is live. In Case (1), by Lemma 4.6.38, $H(u)$ 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 after LINK-COL, SHORTCUT and ALTER all live vertices become finished in the next phase.

In Case (2), u is dormant. Then by Lemma 4.6.44, $|H(u)| \geq b$ with probability at least $1-b^{-1}$. If this event happens, the probability of no leader in $H(u)$ is at most $(1-b^{-2/3})^b \leq \exp(-b^{-1/3}) \leq b^{-1}$. Furthermore, if $|H(u)| \geq b$, then u must own a block of $\Theta(b^{78})$ processors. Thus, we are able to simulate concurrent write on $u.p$ as described in LINK-COL according to Lemma 4.6.34 and the fact that the size of table $|H(u)| \leq b^6$.

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

$$\frac{n' \cdot b^{-1/2}}{n' \cdot b^{-1/4}} \leq b^{-1/4}. \tag{4.4}$$

Removing the assumption

In this section, we remove Assumption 4.6.6.

Recall that we set $b = \delta^{1/84}$ where $\delta = m/n'$. 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 $b = (m/\widetilde{n})^{1/84}$ 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 that a vertex is fully dormant in Lemma 4.6.41. Using $\widetilde{n}$ to rewrite Inequality (4.3), we have:

   $$\frac{n'-1}{m/b^{78}} = \frac{n'-1}{\widetilde{n}b^6} \leq \frac{n'}{n'b^6} = \frac{1}{b^6},$$

   followed from $m/\widetilde{n} = b^{84}$ and $\widetilde{n} \geq n'$. Therefore Lemma 4.6.41 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 (4.4) 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-COL. We give the *update rule* of $\widetilde{n}$:

Update rule of $\widetilde{n}$:

If $m/n \leq \log^c n$ then set $\widetilde{n} \coloneqq n/\log^c n$ for the first phase (after the PREPARE-COL), else set $\widetilde{n} \coloneqq n$.

At the beginning of each phase, update $\widetilde{n} \coloneqq \widetilde{n}/b^{1/4}$ then update $b \coloneqq (m/\widetilde{n})^{1/84}$.

So $b \geq \log^{c/84} n$ is large enough. By the above argument, we immediately have the following:

Lemma 4.6.45. *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, above lemma, and a union bound, the following lemma is immediate:

Lemma 4.6.46. *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 $i \geq 1$.*

Running Time

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

Lemma 4.6.47. *After the* PREPARE-COL*, 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 $n_{i+1} \leq n_i/(m/n_i)^{1/336}$, which gives $m/n_{t+1} \geq (m/n_1)^{(337/336)^t}$. If $t = \lceil \log_{337/336} \log_{m/n_1} m \rceil + 1$ then $n_{t+1} < 1$, which leads to $n' = 0$ at the beginning of phase $t+1$. By Lemma 4.6.2 and monotonicity, the algorithm terminates and outputs the correct connected components in this phase since no parent changes. □

Theorem 4.6.48 (Connectivity)**.** *There is an COLLISION CRCW PRAM algorithm using $O(m)$ processors that computes the connected components of any given graph. With probability $1 - 1/poly(((m+n)\log n)/n)$, it runs in $O(\log(d+1) \log \log_{2(m+n)/n} n)$ time.*

Proof. We set $c = 10^9$ in the PREPARE-COL. If $m/n > \log^c n$, then $\widetilde{n} = n$ in the first phase by Lemma 4.6.37 and the update rule. Since $\delta = m/n_1 \geq \log^c n$, we have that $b \geq \delta^{1/84} \geq \log^{1000} n$ in all phases. By Lemma 4.6.46, $\widetilde{n} \geq n'$ in all phases before phase $\log n$ with probability at least $1 - \log n \cdot b^{-1/4} = 1 - 1/\text{poly}(m \log n/n)$. If this event happens, by Lemma 4.6.47, the number of phases is $O(\log \log_{2(m+n)/n} n)$. By Lemma 4.6.39, the total running time is $O(\log(d+1) \log \log_{2(m+n)/n} n)$ with good probability.

If $m/n \leq \log^c n$, then by Lemma 4.6.37 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 $1 - 1/\text{poly}(m/n_1 \cdot \log n)$, COLLISION CRCW Connected Components algorithm takes time $O(\log(d+1) \log \log_{2(m+n)/n} n)$. Taking a union bound, we obtain that with probability at least $1 - 1/\text{poly}(m/n_1 \cdot \log n) - 1/\log^c n \geq 1 - 1/\text{polylog}(n) = 1 - 1/\text{poly}(m \log n/n)$, the total running time is $O(\log \log n) + O(\log(d+1) \cdot \log \log_{2(m+n)/n_1} n) = O(\log(d+1) \log \log_{2(m+n)/n} n)$. □

Chapter 5: 2-Edge and 2-Vertex Connectivity

In this chapter, we will show how to use graph connectivity and spanning tree algorithms introduced in Chapter 4 to design 2-edge and 2-vertex connectivity algorithms. 2-Edge connectivity and 2-vertex connectivity (biconnectivity) are two fundamental problems in graph theory. Consider an n-vertex, m-edge undirected graph G. A bridge of G is an edge whose removal increases the number of connected components of G. In the 2-edge connectivity problem, the goal is to find all the bridges of G. For any two different edges e, e' of G, e, e' are in the same biconnected component (block) of G if and only if there is a simple cycle which contains both e, e'. If we define a relation R such that eRe' if and only if $e = e'$ or e, e' are contained by a simple cycle, then R is a equivalence relation [23]. Thus, a biconnected component is an induced graph of an equivalence class of R. In the biconnectivity problem, the goal is to output all the biconnected components of G. Our algorithms can be implemented in the MPC model with small number of rounds and small total space.

5.1 Overview of techniques

Biconnectivity: At a high level our biconnectivity algorithm is based on a framework proposed by [50]. The main idea is to reduce the problem of finding biconnected components of G to the problem of finding connected components of G'. At first glance, it should be efficiently solved by the connectivity algorithm (see Chapter 4). However, there are two main issues: 1) since the parallel time of the MPC algorithm described in Chapter 4 depends on the diameter of the input graph, we need to make the diameter of G' small, 2) we need to construct G' efficiently. Let us first consider the first issue, and we will discuss the second issue later.

We give an analysis of the diameter of $G' = (V', E')$ constructed by [50]. Without loss of gen-

erality, we can suppose the input $G = (V, E)$ is connected (otherwise we can first run connectivity algorithm to find all connected components). Each vertex in G' corresponds to an edge of G. Let T be an arbitrary spanning tree of G with depth d. Each non-tree edge e can define a simple cycle C_e which contains the edge e and the unique path between the endpoints of e in the tree T. Thus, the length of C_e is at most $2d + 1$. If there is a such cycle containing any two tree edges $\{u, v\}, \{v, w\}$, vertices $\{u, v\}, \{v, w\}$ are connected in G'. For each non-tree edge e, we connect the vertex e to the vertex e' in graph G' where e' is an arbitrary tree edge in the cycle C_e. By the construction of G', any e, e' from the same connected components of G' should be in the same biconnected components of G. Now consider arbitrary two edges e, e' in the same biconnected component of G. There must be a simple cycle C which contains both edges e, e' in G. Since all the simple cycles defined by the non-tree edges are a cycle basis of G [23], the edge set of C can be represented by the xor sum of all the edge sets of k basis cycles $C_1, C_2, \cdots, C_k$ where C_i is a simple cycle defined by a non-tree edge e_i on the cycle C. k is upper bounded by the bi-diameter of G. Furthermore, we can assume C_i intersects C_{i+1}. There should be a path between e, e' in G', and the length of the path is at most $\sum_{i=1}^{k} |C_i| \leq O(k \cdot d)$. So, the diameter of G' is upper bounded by $O(k \cdot d)$.

Now let us consider how to construct G' efficiently. The bottleneck is to determine whether the tree edges $\{u, v\}, \{v, w\}$ should be connected in G' or not. Suppose w is the parent of v and v is the parent of u. The vertex $\{u, v\}$ should connect to the vertex $\{v, w\}$ in G' if and only if there is a non-tree edge that connects a vertex x in the subtree of u and a vertex y which is on the outside of the subtree of v. For each vertex x, let $\text{lev}(x)$ be the minimum depth of the lowest common ancestor (LCA) of (x, y) over all the non-tree edges $\{x, y\}$. Then $\{u, v\}$ should be connected to $\{v, w\}$ in G' if and only if there is a vertex x in the subtree of u in G such that $\text{lev}(x)$ is smaller than the depth of v. Since the vertices in a subtree should appear consecutively in the DFS sequence, this question can be solved by some range queries over the DFS sequence. Next, we will discuss how to compute the DFS sequence of a tree.

DFS sequence: The DFS sequence of a tree is a variant of the Euler tour representation of the tree. For an n-vertex tree T, [50] gives an $O(\log n)$ parallel time PRAM algorithm for the Euler tour representation of T. However, since their construction method will destroy the tree structure, it is hard to get a faster MPC algorithm based on this framework. In the following, we show the idea of our new algorithm for computing a DFS sequence.

First of all, we use our spanning tree algorithm (see Chapter 4) to compute a rooted tree, reducing the problem to computing a DFS sequence for a rooted tree. The idea is motivated by TeraSort [51]. If the size of the tree is small enough such that it can be handled by a single machine, then we can just use a single machine to generate its DFS sequence. Otherwise, our algorithm can be roughly described as follows. (Recall that δ is the parameter such that each machine has $\Theta(n^{\delta})$ local memory.)

1. Sample $n^{\delta/2}$ leaves $l_1, l_2, \cdots, l_s$.
2. Determine the order of sampled leaves in the DFS sequence.
3. Compute the DFS sequence $\widetilde{A}$ of the tree which only consists of sampled leaves and their ancestors.
4. Compute the DFS sequence A_v of every root-v subtree which does not contain any sampled leaf.
5. Merge $\widetilde{A}$ and all the A_v.

The MPC implementation of the first and second steps go as follows. Since we only sample $n^{\delta/2}$ leaves, we can send them to a single machine. We generate queries for every pair of sampled leaves where each query (l_i, l_j) queries LCA of (l_i, l_j). We have n^{δ} such queries in total. Since the input tree is rooted, we can use a doubling algorithm to preprocess a data structure in $O(\log d)$ parallel time and answer all the queries simultaneously in $O(\log d)$ parallel time. Thus, we know the LCA of any pair of sampled leaves, and we can store this all on a single machine. Based on the information of LCAs of each pair of sampled leaves, we are able to determine the order of the leaves.

For the third step, suppose the sampled leaves have order $l_1, l_2, \cdots, l_s$. Let v be the root of the tree. Then the DFS sequence $\widetilde{A}$ should be: the path from v to l_1, the path from l_1 to the LCA of

(l_1, l_2), the path from the LCA of (l_1, l_2) to l_2, the path from l_2 to the LCA of (l_2, l_3), ..., the path from l_s to v. We can find these paths simultaneously by a doubling algorithm together with a divide-and-conquer algorithm in $O(\log d)$ parallel time.

In the fourth step, we apply the procedure recursively. Suppose the total number of leaves in the tree is $q \leq n$. Since we randomly sampled $n^{\delta/2}$ number of leaves, with high probability, each subtree which does not contain a sampled leaf will have at most $O(q/n^{\delta/2})$ number of leaves. Thus, the depth of the recursion will be at most a constant, $O(1/\delta)$.

Notice that if we simply apply the doubling algorithm to solve the LCA problem and generate multiple path sequences, the total space needed will be $\Omega(n \log d)$ which is not linear in the size of the tree T. We also show how to compress the tree T into a new tree T' which only contains at most $n/\lceil \log d \rceil$ vertices. We argue that applying the doubling algorithm on T' is sufficient for us to find the DFS sequence of T. Thus, our final space usage is linear in the size of the tree.

2**-Edge connectivity:** Without loss of generality, we can assume the input graph G is connected (otherwise, we can first apply the connectivity algorithm to find all connected components, e.g., see Chapter 4). Consider a rooted spanning tree T and an edge $e = \{u, v\}$ in G. Suppose the depth of u is at least the depth of v in T, i.e., v cannot be a child of u. The edge e is not a bridge if and only if either e is a non-tree edge or there is a non-tree edge $\{x, y\}$ connecting the subtree of u and a vertex outside of the subtree of u. Similarly, the second case can be solved by some range queries over the DFS sequence of T.

5.2 DFS sequence of a tree

In this section, we show how to compute a DFS sequence of a rooted tree. We first introduce several subroutines and then we will show how to use these subroutines to build the DFS sequence.

Since we can use our spanning tree algorithm to get a rooted tree, in this section, we only consider how to get a Depth-First-Search (DFS) sequence for a rooted tree. Before we go to the details, let us firstly give formal definitions of some useful concepts.

Definition 5.2.1 (Children in the forest)**.** *Given a set of parent pointers (See Definition 4.2.5)* $\mathrm{par} : V \to V$ *on a vertex set* V. $\forall u, v \in V, u \neq v$ *if* $\mathrm{par}(u) = v$, *then we say* u *is a child of* v. $\forall v \in V$, *we can define* $\mathrm{child}_{\mathrm{par}}(v)$ *as the set of all children of* v, *i.e.* $\mathrm{child}_{\mathrm{par}}(v) = \{u \in V \mid u \neq v, \mathrm{par}(u) = v\}$. *Furthermore, if* u *is the* k^{th} *smallest vertex in the children set* $\mathrm{child}_{\mathrm{par}}(v)$, *then we say* $\mathrm{rank}_{\mathrm{par}}(u) = k$, *or* u *is the* k^{th} *child of* v. *If* $\mathrm{par}(v) = v$, *then* $\mathrm{rank}_{\mathrm{par}}(v) = 1$. *We use* $\mathrm{child}_{\mathrm{par}}(v, k)$ *to denote the* k^{th} *child of* v.

For simplicity of the notation, if $\mathrm{par} : V \to V$ is clear in the context, we just use $\mathrm{child}(v), \mathrm{rank}(v)$ and $\mathrm{child}(v, k)$ to denote $\mathrm{child}_{\mathrm{par}}(v), \mathrm{rank}_{\mathrm{par}}(v)$ and $\mathrm{child}_{\mathrm{par}}(v, k)$ respectively.

Definition 5.2.2 (Leaves in the forest)**.** *Given a set of parent pointers (See Definition 4.2.5)* $\mathrm{par} : V \to V$ *on a vertex set* V. *If* $\mathrm{child}_{\mathrm{par}}(v) = \emptyset$, *then* v *is called a leaf. The set of all the leaves of* par *is defined as* $\mathrm{leaves}(\mathrm{par}) = \{v \mid \mathrm{child}_{\mathrm{par}}(v) = \emptyset\}$.

Definition 5.2.3 (Subtree)**.** *Let* $\mathrm{par} : V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V. *Let* $v \in V, V' = \{u \in V \mid v$ *is an ancestor (Definition 5.2.8) of* $u\}$. *Let* $\mathrm{par}' : V' \to V'$ *be a set of parent pointers on* V'. *If* $\forall u \in V' \setminus \{v\}, \mathrm{par}'(u) = \mathrm{par}(u)$, *and* $\mathrm{par}'(v) = v$, *then we say* par′ *is the subtree of* v *in* par. *For* $u \in V'$, *we say* u *is in the subtree of* v.

Definition 5.2.4 (Depth-First-Search (DFS) sequence)**.** *Let* $\mathrm{par} : V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V. *Let* v *be a vertex in* V. *If* v *is a leaf (See Definition 5.2.2) in* par, *then the DFS sequence of the subtree (See Definition 5.2.3) of* v *is* (v). *Otherwise the DFS sequence of the subtree of* v *in* par *is recursively defined as*

$$(v, a_{1,1}, a_{1,2}, \cdots, a_{1,n_1}, v, a_{2,1}, a_{2,2}, \cdots, a_{2,n_2}, v, \cdots, a_{k,1}, a_{k,2}, \cdots, a_{k,n_k}, v),$$

where $k = |\mathrm{child}(v)|$ *is the number of children (See Definition 5.2.1) of* v, *and* $\forall i \in [k], (a_{i,1}, \cdots, a_{i,n_i})$ *is the DFS sequence of the subtree of* $\mathrm{child}(v, i)$, *i.e. the* i^{th} *child of* v.

If $\forall u \in V, \mathrm{par}^{(\infty)}(u) = v$, *then the subtree of* v *is exactly* par, *and thus the DFS sequence of the subtree of* v *is also called the DFS sequence of* par .

Here are some useful facts of the above defined DFS sequence.

Fact 5.2.5. *Let* par $: V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V*, and* par *has a unique root. Let* $A = (a_1, a_2, \cdots, a_m)$ *be the DFS sequence (See Definition 5.2.4) of* par . *Then, A satisfies the following properties:*

1. $\forall v \in V$, v *appears exactly* $|\text{child}(v)| + 1$ *times in A.*
2. *If* a_i *is the* k^{th} *time that* v *appears, and* a_j *is the* $(k+1)^{th}$ *time that* v *appears. Then* $(a_{i+1}, a_{i+2}, \cdots, a_{j-1})$ *is the DFS sequence of the subtree of* child(v, k) *(See Definition 5.2.1), the* k^{th} *child of* v*. Furthermore,* a_{i+1} *is the first time that* child(v, k) *appears, and* a_{j-1} *is the last time of* child(v, k) *appears.*
3. *If* a_i *is the first time that* v *appears, and* a_j *is the last time that* v *appears. Then* $(a_i, a_{i+1}, \cdots, a_j)$ *is the DFS sequence of the subtree of* v*.*
4. $m = 2|V| - 1$.

Proof. The property 1, 2, 3 directly follows by Definition 5.2.4.

For property 4, notice that $\forall u \in V, \text{par}(u) \neq u, u$ can only be a child of par(u). Thus, $\sum_{v \in V}(|\text{child}(v)| + 1) = |V| - 1 + |V| = 2|V| - 1$. □

Due to the above fact, if v is a leaf in par, then it will only once in the DFS sequence. Thus, we are able to determine the order of all the leaves.

Definition 5.2.6 (The order of the leaves)**.** *Let* par $: V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V*, and* par *has a unique root. Let* $A = (a_1, a_2, \cdots, a_m)$ *be the DFS sequence (See Definition 5.2.4) of* par . *Let* u, v *be two leaves (See definition 5.2.2) of* par . *If* u *appears before* v *in A, then we say* $u <_{\text{par}} v$.

5.2.1 Compressed rooted tree

Given a set of parent pointers par $: V \to V$, we will show how to compress the rooted tree represented by par.

Algorithm 12 Construction of a Compressed Rooted Tree

1: **procedure** COMPRESS(par : $V \to V$) ▷ par : $V \to V$ is a set of parent pointers representing a rooted tree on a set V of n vertices (par has a unique root r).
2: Compute the depth of par, the depth of each vertex and set $d \leftarrow \text{dep}(\text{par})$, $t \leftarrow \lceil \log d \rceil$. ▷ Run TREECONTRACTION($(V, \emptyset)$, par) (see Algorithm 3). Let the output $g^{(r)} : V \to \mathbb{Z}_{\geq 0}$ be $\text{dep}_{\text{par}} : V \to \mathbb{Z}_{\geq 0}$ (see Lemma 4.2.9).
3: Compute $\text{par}^{(t)}(u)$ for each $u \in V$.
4: $V' \leftarrow \{v \in V \mid \text{dep}_{\text{par}}(v) \bmod t = 0, \exists u \in V, v = \text{par}^{(t)}(u)\}$.
5: Initialize $\text{par}' : V' \to V'$. For each $v \in V'$, $\text{par}'(v) \leftarrow \text{par}^{(t)}(v)$.
6: Output V', par′.
7: **end procedure**

Lemma 5.2.7 (Properties of a compressed rooted tree)**.** *Let* par $: V \to V$ *be a set of parent pointers on a vertex set* V *with* $|V| > 1$*, and* par *has a unique root. Let* $t = \lceil \log(\text{dep}(\text{par})) \rceil$ *and let* (V', par') =COMPRESS(par) *(see Algorithm 12). Then it has the following properties:*

1. $|V'| \leq \max(|V|/\log(\text{dep}(\text{par})), 1)$.
2. $\forall v \in V', i \in \mathbb{N}, \text{par}'^{(i)}(v) = \text{par}^{(i \cdot t)}(v) \in V'$.
3. $\forall v \in V, \exists i \in \{0, 1, \cdots, 2t\}$, *such that* $\text{par}^{(i)}(v) \in V'$.
4. $\forall v \in V, \forall l \in \mathbb{Z}_{\geq 0}$, *if* $l \in [t, \text{dep}_{\text{par}}(v)]$ *and* $(\text{dep}_{\text{par}}(v) - l) \bmod t = 0$, *then* $\text{par}^{(l)}(v) \in V'$.

Proof. Consider the first property. Consider the case if $\text{dep}(\text{par}) < t$, we know that $V' = \{\text{the root of par}\}$. Next consider the case $\text{dep}(\text{par}) \geq t$. For each $v \in V'$, we define a set

$$S(v) = \{u \in V \mid \text{dep}_{\text{par}}(u) > \text{dep}_{\text{par}}(v), \exists i \in [t-1], \text{par}^{(i)}(u) = v\}.$$

$\forall u \in S(v)$, we have $\text{dep}_{\text{par}}(u) - \text{dep}_{\text{par}}(v) < t$. Since $\forall v \in V'$, $\text{dep}_{\text{par}}(v) \bmod t = 0$, we have $S(v) \cap V' = \emptyset$. Furthermore, it is easy to show that $\forall u \neq v \in V'$, $S(u) \cap S(v) = \emptyset$. Thus, $|V'| + \sum_{v \in V'} |S(v)| = \sum_{v \in V'} (|S(v)| + 1) \leq |V|$. On the other hand, since $\forall v \in V', \exists u \in V, \text{par}^{(t)}(u) = v$, we know that $|S(v)| \geq t - 1$ if v is not the root. If v is the root, since $\text{dep}(\text{par}) \geq t$, we also have $S(v) \geq t - 1$. Therefore $\sum_{v \in V'} (|S(v)| + 1) \geq |V'| \cdot t$. To conclude, $|V'| \leq |V|/t \leq |V|/\log(\text{dep}(\text{par}))$.

Consider the second property. If v is a root vertex, $\text{par}'(v) = \text{par}^{(t)}(v) = v \in V'$. For a non-root vertex $v \in V'$, $\text{dep}_{\text{par}}(\text{par}^{(t)}(v)) = \text{dep}_{\text{par}}(v) - t$. Since $\text{dep}_{\text{par}}(v) \bmod t = 0$, we have

$\text{dep}_{\text{par}}(\text{par}^{(t)}(v)) \mod t = 0$ which means that $\text{par}'(v) = \text{par}^{(t)}(v) \in V'$. Now we prove by induction. Suppose $\text{par}'^{(i-1)}(v) = \text{par}^{((i-1)\cdot t)}(v)$, then $\text{par}'^{(i)}(v) = \text{par}'(\text{par}'^{(i-1)}(v)) = \text{par}^{(t)}(\text{par}^{((i-1)\cdot t)}(v)) = \text{par}^{(i\cdot t)}(v)$.

Consider the third property. For $v \in V$, $\exists j \in \{0, 1, \cdots, t-1\}$, such that $\text{dep}_{\text{par}}(\text{par}^{(j)}(v)) \mod t = 0$. Since $\text{dep}_{\text{par}}(\text{par}^{(j+t)}(v)) \mod t = 0$ and $\text{par}^{(t)}(\text{par}^{(j)}(v)) = \text{par}^{(j+t)}(v)$, we know that $\text{par}^{(j+t)}(v) \in V'$. Since $j + t \leq 2t$, the property holds.

Consider the forth property. For $v \in V$, if $l \leq \text{dep}_{\text{par}}(v)$ and $(\text{dep}_{\text{par}}(v) - l) \mod t = 0$, we have $\text{dep}_{\text{par}}(\text{par}^{(l)}(v)) \mod t = 0$. Since $l \geq t$, we know that $\text{par}^{(t)}(\text{par}^{(l-t)}(v)) = \text{par}^{(l)}(v)$. According to the construction of V', we know that $\text{par}^{(l)}(v) \in V'$. □

5.2.2 Lowest common ancestor

Definition 5.2.8 (Ancestor)**.** *Let* $\text{par} : V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V*. For* $u, v \in V$*, if* $\exists k \in \mathbb{Z}_{\geq 0}$ *such that* $u = \text{par}^{(k)}(v)$*, then* u *is an ancestor of* v*.*

Definition 5.2.9 (Common ancestor and the lowest common ancestor)**.** $\text{par} : V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V*. For* $u, v \in V$*, if* w *is an ancestor of* u *and is also an ancestor of* v*, then* w *is a common ancestor of* (u, v)*. If a common ancestor* w *of* (u, v) *satisfies* $\text{dep}_{\text{par}}(w) \geq \text{dep}_{\text{par}}(x)$ *for any common ancestor* x *of* (u, v)*, then* w *is the lowest common ancestor (LCA) of* (u, v)*.*

Given a rooted tree represented by a set of parent pointers $\text{par} : V \to V$ on a vertex set V, and a set of q queries $Q = \{(u_1, v_1), (u_2, v_2), \cdots, (u_q, v_q)\}$ where $\forall i \in [q], u_i \neq v_i, u_i, v_i \in \text{leaves(par)}$, we show a space efficient algorithm which can output the LCA of each queried pair of vertices. Notice that the assumption that queries only contain leaves is without loss of generality: we can attach an additional child vertex v to each non-leaf vertex u. Thus, v is a leaf vertex. When a query contains u, we can use v to replace u in the query, and the result will not change.

We first show an algorithm (Algorithm 13) with slightly larger space, then we show how to get a space-efficient algorithm in Algorithm 14.

Algorithm 13 Lowest Common Ancestor (larger space)

procedure LCALARGE(par : $V \to V, Q = \{(u_1, v_1), (u_2, v_2), \cdots, (u_q, v_q)\}$) ▷ Lemma 5.2.10.
  Output: lca : $Q \to V \times V \times V$
  $(r, \mathrm{dep}_{\mathrm{par}}, \{g_i \mid i \in \{0\} \cup [r]\}) \leftarrow$ FINDANCESTORS(par). ▷ Algorithm 6.
  Let $Q' \leftarrow \emptyset$.
  $\forall (u, v) \in Q$, if $\mathrm{dep}_{\mathrm{par}}(u) \geq \mathrm{dep}_{\mathrm{par}}(v)$, then let $Q' \leftarrow Q' \cup \{(u, v)\}$; Otherwise let $Q' \leftarrow Q' \cup \{(v, u)\}$.
  Let $h_r : Q' \to Q'$ be an identity mapping.
  for $i = r - 1 \to 0$ **do** ▷ Move u to the almost same depth as v.
    For each $(u, v) \in Q'$, let $(x, v) \leftarrow h_{i+1}(u, v)$. If $\mathrm{dep}_{\mathrm{par}}(x) - 2^i \geq \mathrm{dep}_{\mathrm{par}}(y)$, then let $h_i(u, v) \leftarrow (g_i(x), v)$; Otherwise let $h_i(u, v) \leftarrow (x, v)$.
  end for
  For each $(u, v) \in Q'$, let $h'_r(u, v) \leftarrow h_0(u, v)$.
  for $i = r - 1 \to 0$ **do** ▷ Move u, v to the lowest common ancestor.
    For each $(u, v) \in Q'$, let $(x, y) \leftarrow h'_{i+1}(u, v)$. If $g_i(x) \neq g_i(y)$, then let $h'_i(u, v) \leftarrow (g_i(x), g_i(y)), h'_i(u) \leftarrow g_i(x), h'_i(v) \leftarrow g_i(y)$; Otherwise let $h'_i(u, v) \leftarrow (x, y), h'_i(u) \leftarrow x, h'_i(v) \leftarrow y$.
  end for
  For each $(u, v) \in Q'$, if $(u, v) \in Q$, then let $\mathrm{lca}(u, v) \leftarrow (\mathrm{par}(h'_0(u)), h'_0(u), h'_0(v))$; Otherwise $\mathrm{lca}(v, u) \leftarrow (\mathrm{par}(h'_0(v)), h'_0(v), h'_0(u))$.
  return lca .
end procedure

Lemma 5.2.10. *Let* par : $V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V. *Suppose* par *has a unique root. Let* $Q = \{(u_1, v_1), (u_2, v_2), \cdots, (u_q, v_q)\}$ *be a set of* q *pairs of vertices, and* $\forall i \in [q], u_i \neq v_i$, *neither* u_i *nor* v_i *is the LCA of* (u_i, v_i). *Let* $\mathrm{lca} = \mathrm{LCALARGE}(\mathrm{par}, Q)$ *(Algorithm 13). Then for any* $(u, v) \in Q$, $(p, p_u, p_v) = \mathrm{lca}(u, v)$ *satisfies the following properties:* p *is the lowest common ancestor of* (u, v), $p_u \neq p$ *is an ancestor of* u, $p_v \neq p$ *is an ancestor of* v, *and* $\mathrm{par}(p_u) = \mathrm{par}(p_v) = p$.

Proof. According to Lemma 4.3.6, r should be at most $\lceil \log(\mathrm{dep}(\mathrm{par}) + 1) \rceil$, $\mathrm{dep}_{\mathrm{par}} : V \to \mathbb{Z}_{\geq 0}$ records the depth of every vertex in V, and $\forall i \in \{0\} \cup [r], v \in V$ $g_i(v) = \mathrm{par}^{(2^i)}(v)$.

Then for all $(u, v) \in Q$, either $(u, v) \in Q'$ or $(v, u) \in Q'$. For each $(u, v) \in Q'$, we have $\mathrm{dep}_{\mathrm{par}}(u) \geq \mathrm{dep}_{\mathrm{par}}(v)$. For all $(u, v) \in Q'$, by induction we can prove that $\forall i \in \{0\} \cup [r - 1], (x, y) = h_i(u, v)$ satisfies that x is an ancestor of u, $y = v$, $\mathrm{dep}_{\mathrm{par}}(x) \geq \mathrm{dep}_{\mathrm{par}}(v)$ and $\mathrm{dep}_{\mathrm{par}}(\mathrm{par}^{(2^i)}(x)) < \mathrm{dep}_{\mathrm{par}}(v)$. Thus, $\forall (u, v) \in Q', (x, y) = h_0(u, v)$ satisfies that $\mathrm{dep}_{\mathrm{par}}(x) = \mathrm{dep}_{\mathrm{par}}(y)$ and x is an ancestor of u, $y = v$.

Now let $(u, v) \in Q'$. We have $\mathrm{dep}_{\mathrm{par}}(h'_r(u)) = \mathrm{dep}_{\mathrm{par}}(h'_r(v)), h'_r(u) \neq h'_r(v)$, and $h'_r(u), h'_r(v)$ are

ancestors of u, v respectively. We can prove by induction to get $\forall i \in \{0\} \cup [r], h'_i(u) \neq h'_i(v)$ and $\mathrm{par}^{(2^i)}(h'_i(u)) = \mathrm{par}^{(2^i)}(h'_i(v))$ is a common ancestor of (u, v). Thus, $p = \mathrm{par}(h'_0(u)) = \mathrm{par}(h'_0(v))$ is the lowest common ancestor of (u, v), and $\mathrm{dep}_{\mathrm{par}}(h'_0(u)) = \mathrm{dep}_{\mathrm{par}}(h'_0(v)) = \mathrm{dep}_{\mathrm{par}}(p) + 1$. Since $p_u = h'_0(u), p_v = h'_0(v)$, we complete the proof. □

Algorithm 14 Lowest Common Ancestor (space efficient)

1: **procedure** LCA($\mathrm{par} : V \to V, Q$) ▷ $\mathrm{par} : V \to V$ is a set of parent pointers representing a rooted tree on a set V of n vertices (par has a unique root r), and Q is a set of q queries $Q = \{(u_1, v_1), (u_2, v_2), \cdots, (u_q, v_q)\}$ where $\forall i \in [q], u_i \neq v_i, u_i, v_i \in \mathrm{leaves}(\mathrm{par})$.
2: $(V', \mathrm{par}') \leftarrow$COMPRESS(par). ▷ Algorithm 12.
3: Compute the depth of each vertex in par. ▷ Run TREECONTRACTION($(V, \emptyset), \mathrm{par}$) (see Algorithm 3). Let the output $g^{(r)} : V \to \mathbb{Z}_{\geq 0}$ be $\mathrm{dep}_{\mathrm{par}} : V \to \mathbb{Z}_{\geq 0}$ (see Lemma 4.2.9).
4: Compute the depth of each vertex in par′. ▷ Run TREECONTRACTION($(V', \emptyset), \mathrm{par}'$) (see Algorithm 3). Let the output $g^{(r)} : V' \to \mathbb{Z}_{\geq 0}$ be $\mathrm{dep}_{\mathrm{par}'} : V \to \mathbb{Z}_{\geq 0}$ (see Lemma 4.2.9).
5: Set $d \leftarrow \mathrm{dep}(\mathrm{par}), t \leftarrow \lceil \log d \rceil$.
6: Compute mappings $g_0, g_1, \cdots g_t : V' \to V'$ such that $\forall v \in V', j \in \{0, 1, \cdots, t\}$, $g_j(v) = \mathrm{par}'^{(2^j)}(v)$. ▷ Run FINDANCESTORS(par) (see Algorithm 6).
7: **for** $(u_i, v_i) \in Q$ **do** ▷ Suppose $\mathrm{dep}_{\mathrm{par}}(u_i) \geq \mathrm{dep}_{\mathrm{par}}(v_i)$.
8: If $\mathrm{dep}_{\mathrm{par}}(u_i) > \mathrm{dep}_{\mathrm{par}}(v_i)+2t$, find an ancestor $\widehat{u_i}$ of u_i in par such that $\mathrm{dep}_{\mathrm{par}}(\widehat{u_i}) \leq \mathrm{dep}_{\mathrm{par}}(v_i)+2t$ and $\mathrm{dep}_{\mathrm{par}}(\widehat{u_i}) \geq \mathrm{dep}_{\mathrm{par}}(v_i)$. Otherwise, $\widehat{u_i} \leftarrow u_i$.
9: If $\exists j \in [4t]$ $\mathrm{par}^{(j)}(\widehat{u_i})$ is the LCA of $(\widehat{u_i}, v_i)$ in par, set $\mathrm{lca}(u_i, v_i) = (\mathrm{par}^{(j)}(\widehat{u_i}), x, y)$ where x, y are children of $\mathrm{par}^{(j)}(\widehat{u_i})$ and x, y are ancestors of $\widehat{u_i}, v_i$ respectively. The query of (u_i, v_i) is finished.
10: Find an ancestor u'_i of $\widehat{u_i}$ in par such that u'_i is the closest vertex to $\widehat{u_i}$ in V', i.e., $\mathrm{dep}_{\mathrm{par}}(\widehat{u_i}) - \mathrm{dep}_{\mathrm{par}}(u'_i)$ is minimized. Similarly, find an ancestor v'_i of v_i in par such that v'_i is the closest vertex to v_i in V', i.e., $\mathrm{dep}_{\mathrm{par}}(v_i) - \mathrm{dep}_{\mathrm{par}}(v'_i)$ is minimized.
11: Find $u''_i \neq v''_i \in V'$ such that they are ancestors of u'_i and v'_i respectively, and $\mathrm{par}'(u''_i) = \mathrm{par}'(v''_i)$ is the LCA of (u'_i, v'_i) in par′. ▷ Run LCALARGE(par', Q') for Q' which contains all (u'_i, v'_i) (see Algorithm 13).
12: Find the smallest $j \in [2t]$ such that $\mathrm{par}^{(j)}(u''_i) = \mathrm{par}^{(j)}(v''_i)$. Set $\mathrm{lca}(u_i, v_i) = (\mathrm{par}^{(j)}(u''_i), \mathrm{par}^{(j-1)}(u''_i), \mathrm{par}^{(j-1)}(v''_i))$.
13: **end for**
14: **end procedure**

Remark 5.2.11. *Before we analyze the algorithm* $\mathrm{LCA}(\mathrm{par}, Q)$ *(Algorithm 14), let us discuss the details of the implementation of line 8 of the algorithm. To implement line 8, we firstly check whether* $\mathrm{dep}_{\mathrm{par}}(u_i) > \mathrm{dep}_{\mathrm{par}}(v_i) + 2t$. *If it is not true, we can set* $\widehat{u_i}$ *to be* u_i *directly. Otherwise, according to Lemma 5.2.7, there is a* $j \in \{0, 1, \cdots, 2t\}$ *such that* $\mathrm{par}^{(j)}(u_i) \in V'$. *Since* $\mathrm{dep}_{\mathrm{par}}(u_i) > \mathrm{dep}_{\mathrm{par}}(v_i) + 2t$, $\mathrm{dep}_{\mathrm{par}}(\mathrm{par}^{(j)}(u_i)) > \mathrm{dep}_{\mathrm{par}}(v_i)$. *We initialize* $\widehat{u_i}$ *to be* $\mathrm{par}^{(j)}(u_i) \in V'$. *For* $k = t \to 0$, *if* $\mathrm{dep}_{\mathrm{par}}(g_k(\widehat{u_i})) > \mathrm{dep}_{\mathrm{par}}(v_i)$ *(i.e.,* $\mathrm{dep}_{\mathrm{par}}(\mathrm{par}'^{(2^k)}(\widehat{u_i})) > \mathrm{dep}_{\mathrm{par}}(v_i)$*), we set* $\widehat{u_i} \leftarrow$

$g_k(\widehat{u_i}) = \mathrm{par}'^{(2^k)}(\widehat{u_i})$. *Due to Lemma 5.2.7 again, the final* $\widehat{u_i}$ *must satisfy* $\mathrm{dep}_{\mathrm{par}}(\widehat{u_i}) \geq \mathrm{dep}_{\mathrm{par}}(v_i)$ *and* $\mathrm{dep}_{\mathrm{par}}(\widehat{u_i}) \leq \mathrm{dep}_{\mathrm{par}}(v_i) + 2t$. *This step takes time* $O(t)$.

Lemma 5.2.12 (LCA algorithm)**.** *Let* par : $V \to V$ *be a set of parent pointers on a vertex set* V. par *has a unique root. Let* $Q = \{(u_1, v_1), (u_2, v_2), \cdots, (u_q, v_q)\}$ *be a set of* q *pairs of vertices where* $\forall i \in [q], u_i \neq v_i, u_i, v_i \in \mathrm{leaves}(\mathrm{par})$. *Let* $\mathrm{lca} : Q \to V \times V \times V$ *be the output of* LCA(par) *(see Algorithm 14). For* $(u_i, v_i) \in Q$, $(p_i, p_{i,u_i}, p_{i,v_i}) = \mathrm{lca}(u_i, v_i)$ *satisfies that* p_i *is the LCA of* (u_i, v_i), p_{i,u_i}, p_{i,v_i} *are ancestors of* u_i, v_i *respectively, and* p_{i,u_i}, p_{i,v_i} *are children of* p_i.

Proof. Without loss of generality, we can assume $\mathrm{dep}_{\mathrm{par}}(u_i) \geq \mathrm{dep}_{\mathrm{par}}(v_i)$. After line 8, $\widehat{u_i}$ satisfies $\mathrm{dep}_{\mathrm{par}}(\widehat{u_i}) \geq \mathrm{dep}_{\mathrm{par}}(v_i)$ and $\mathrm{dep}_{\mathrm{par}}(\widehat{u_i}) \leq \mathrm{dep}_{\mathrm{par}}(v_i) + 2t$. Notice that the LCA of (u_i, v_i) in par is the same as the LCA of $(\widehat{u_i}, v_i)$ in par. In line 9, if we find the LCA of $(\widehat{u_i}, v_i)$, then the lemma holds for $\mathrm{lca}(u_i, v_i)$. Otherwise, the depth of the LCA of $(\widehat{u_i}, v_i)$ is smaller than $\mathrm{dep}_{\mathrm{par}}(\widehat{u_i}) - 4t \leq \mathrm{dep}_{\mathrm{par}}(v_i) - 2t$. By combining with Lemma 5.2.7, neither of u_i' nor v_i' in line 10 can be the LCA of $(\widehat{u_i}, v_i)$ in par. Thus, the LCA of (u_i, v_i) in par is the same as the LCA of (u_i', v_i') in par. According to line 11, u_i'', v_i'' are ancestors of u_i', v_i' respectively in both par and par′, but neither of u_i'' nor v_i'' is the common ancestor of (u_i', v_i'). Furthermore, $\mathrm{par}'(u_i'') = \mathrm{par}'(v_i'')$ is the LCA of u_i', v_i' in par′. Thus, $\mathrm{par}'(u_i'')$ is a common ancestor of (u_i', v_i') in par. According to Lemma 5.2.10, line 11 can be implemented by Algorithm 13. By combining with Lemma 5.2.7, we know that there exists $j \in [2t]$ such that $\mathrm{par}^{(j)}(u_i'')$ is the LCA of (u_i', v_i') in par. In line 12, we can find the LCA of (u_i', v_i') in par and thus the LCA of (u_i, v_i). □

5.2.3 Multi-paths generation

Definition 5.2.13 (Path between two vertices)**.** par : $V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V. *For* $u, v \in V$, *if* $\mathrm{par}^{(\infty)}(u) = \mathrm{par}^{(\infty)}(v)$, *then the path from* u *to* v *is a sequence* $(x_1, x_2, \cdots, x_j, x_{j+1}, \cdots x_k)$ *such that* $\forall i \neq i' \in [k], x_i \neq x_{i'}, x_1 = u, x_k = v, x_j$ *is the lowest common ancestor of* $(u, v), \forall i \in [j-1], \mathrm{par}(x_i) = x_{i+1}$, *and* $\forall i \in \{j+1, j+2, \cdots, k\}, \mathrm{par}(x_i) = x_{i-1}$.

Consider a rooted tree represented by a set of parent pointers $\text{par} : V \to V$ on a vertex set V and a set of q vertex-ancestor pairs $Q = \{(u_1, v_1), (u_2, v_2), \cdots, (u_q, v_q)\}$ where $\forall i \in [q]$, v_i is an ancestor of u_i.

We first show an algorithm (Algorithm 15) with slightly larger space. Then we show how to improve the space usage. The following lemma claims the properties of the outputs of Algorithm 15.

Algorithm 15 Multiple-Paths Generation (larger space)

procedure MULTIPATHLARGE(par : $V \to V, Q = \{(u_1, v_1), (u_2, v_2), \cdots, (u_q, v_q)\}$) ▷ Lemma 5.2.14.
  Output: $\text{dep}_{\text{par}} : V \to \mathbb{Z}_{\geq 0}, \{P_i \subseteq V \mid i \in [q]\}$.
  $(r, \text{dep}_{\text{par}}, \{g_i \mid i \in \{0\} \cup [r]\}) \leftarrow$ FINDANCESTORS(par). ▷ Algorithm 6.
  $\forall j \in [q]$, let $S_j^{(0)} \leftarrow \{(u_j, v_j) \mid (u_j, v_j) \in Q\}$.
  for $i = 1 \to r$ **do**
    for $j = 1 \to q$ **do** ▷ $S_j^{(i)}$ is a set of segments partitioned the path from u_j to v_j.
      Let $S_j^{(i)} \leftarrow \emptyset$.
      for $(x, y) \in S_j^{(i-1)}$ **do**
        if $\text{dep}_{\text{par}}(x) - \text{dep}_{\text{par}}(y) > 2^{r-i}$ **then** $S_j^{(i)} \leftarrow S_j^{(i)} \cup \{(x, g_{r-i}(x)), (g_{r-i}(x), y)\}$.
        else $S_j^{(i)} \leftarrow S_j^{(i)} \cup \{(x, y)\}$.
        end if
      end for
    end for
  end for ▷ $S_j^{(r)}$ only contains segments with length 1.
  Let $\forall j \in [q], P_j \leftarrow \{u_j\}$.
  for $j = 1 \to q$ **do**
    for $(x, y) \in S_j^{(r)}$ **do**
      Let $P_j \leftarrow P_j \cup \{y\}$.
    end for
  end for
end procedure

And the proof is similar to the proof of Lemma 4.3.8.

Lemma 5.2.14. *Let* $\text{par} : V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V. *Let* $Q = \{(u_1, v_1), (u_2, v_2), \cdots, (u_q, v_q)\} \subseteq V \times V$ *satisfy* $\forall j \in [q]$, v_j *is an ancestor (See Definition 5.2.8) of* u_j *in* par. *Let* $(\text{dep}_{\text{par}}, \{P_j \mid j \in [q]\}) = \text{MULTIPATHLARGE}(\text{par}, Q)$ *(Algorithm 15). Then* $\text{dep}_{\text{par}} : V \to \mathbb{Z}_{\geq 0}$ *records the depth of every vertex in* V *and* $\forall j \in [q], P_j \subseteq V$ *is the set of all vertices on the path from* u_j *to* v_j, *i.e.* $P_j = \{v \in V \mid \exists k_1, k_2 \in \mathbb{Z}_{\geq 0}, v = \text{par}^{(k_1)}(u_j), v_j = \text{par}^{(k_2)}(v)\}$. *Furthermore,* r *should be at most* $\lceil \log(\text{dep}(\text{par}) + 1) \rceil$.

Proof. By Lemma 4.3.6, since $(r, \mathrm{dep}_{\mathrm{par}}, \{g_i \mid i \in \{0\} \cup [r]\}) = \textsc{FindAncestors}(\mathrm{par})$, we know r should be at most $\lceil \log(\mathrm{dep}(\mathrm{par}) + 1) \rceil$, $\mathrm{dep}_{\mathrm{par}} : V \to \mathbb{Z}_{\geq 0}$ records the depth of every vertex in V, and $\forall i \in \{0\} \cup [r], v \in V\ g_i(v) = \mathrm{par}^{(2^i)}(v)$.

For $j \in [q]$, let us prove that P_j is the vertex set of all the vertices on the path from u_j to its ancestor v_j. We use divide-and-conquer to get P_j. The following claim shows that $S_j^{(i)}$ is a set of segments which is a partition of the path from u_j to v_j, and each segment has length at most 2^{r-i}.

Claim 5.2.15. *$\forall i \in \{0\} \cup [r], j \in [q]$ $S_j^{(i)}$ satisfies the following properties:*

1. *$\exists (x, y) \in S_j^{(i)}$ such that $x = u_j$.*
2. *$\exists (x, y) \in S_j^{(i)}$ such that $y = v_j$.*
3. *$\forall (x, y) \in S_j^{(i)}, \mathrm{dep}_{\mathrm{par}}(y) - \mathrm{dep}_{\mathrm{par}}(x) \leq 2^{r-i}$.*
4. *$\forall (x, y) \in S_j^{(i)}$, if $y \neq v_j$, then $\exists (x', y') \in S_j^{(i)}, x' = y$.*
5. *$\forall (x, y) \in S_j^{(i)}, \exists k \in \mathbb{Z}_{\geq 0}, \mathrm{par}^{(k)}(x) = y$.*

Proof. We fix a $j \in [q]$. Our proof is by induction. According to line 4, all the properties hold when $i = 0$. Suppose all the properties hold for $i - 1$. For property 1, by induction we know there exists $(x, y) \in S_j^{(i-1)}$ such that $x = u_j$. Then by line 9 and line 10, there must be an (x, y') in $S_j^{(i)}$. For property 2, by induction we know there exists $(x, y) \in S_j^{(i-1)}$ such that $y = v_j$. Thus, there must be an (x', y) in $S_j^{(i)}$. For property 3, if (x, y) is added into $S_j^{(i)}$ by line 10, then $\mathrm{dep}_{\mathrm{par}}(x) - \mathrm{dep}_{\mathrm{par}}(y) \leq 2^{r-i}$. Otherwise, in line 9, we have $\mathrm{dep}_{\mathrm{par}}(x) - \mathrm{dep}_{\mathrm{par}}(g_{r-i}(x)) \leq 2^{r-i}, \mathrm{dep}_{\mathrm{par}}(g_{r-i}(x)) - \mathrm{dep}_{\mathrm{par}}(y) \leq 2^{r-i+1} - 2^{r-i} = 2^{r-i}$. For property 4, if (x, y) is added into $S_j^{(i)}$ by line 10, then by induction there is $(y, y') \in S_j^{(i-1)}$, and thus by line 10 and line 9, there must be $(y, y'') \in S_j^{(i)}$. Otherwise, in line 9 will generate two pairs $(x, g_{r-i}(x)), (g_{r-i}(x), y)$. For $(x, g_{r-i}(x))$, the property holds. For $(g_{r-i}(x), y)$, there must be $(y, y') \in S_{i-1}$ and thus there should be $(y, y'') \in S^{(i)}$. For property 5, since $g_{r-i}(x) = \mathrm{par}^{(r-i)}(x)$, for all pairs generated by line 9 and line 10, the property holds. □

By Claim 5.2.15, we know

$$
\begin{aligned}
S_j^{(r)} = \{ & \\
& (u_j, \mathrm{par}(u_j)), \\
& \left(\mathrm{par}(u_j), \mathrm{par}^{(2)}(u_j)\right), \\
& \left(\mathrm{par}^{(2)}(u_j), \mathrm{par}^{(3)}(u_j)\right), \\
& \cdots, \\
& \left(\mathrm{par}^{(\mathrm{dep}_{\mathrm{par}}(u_j)-\mathrm{dep}_{\mathrm{par}}(v_j)-1)}(u_j), \mathrm{par}^{(\mathrm{dep}_{\mathrm{par}}(u_j)-\mathrm{dep}_{\mathrm{par}}(v_j))}(u_j)\right) \\
\} & .
\end{aligned}
$$

Thus, P_j is the set of all the vertices on the path from u_j to an ancestor v_j. □

Then we show a space efficient algorithm MULTIPATH(par, Q) (see Algorithm 16) which can generate all the paths $P(u_1, v_1), P(u_2, v_2), \cdots, P(u_q, v_q)$.

Remark 5.2.16. *Before we analyze the correctness of Algorithm 16, let us discuss some details. In line 8, if the length of the path is at most* $2t$*, then we can generate the path in* $O(t)$ *rounds. In the* j*-th round, we can find the vertex* $\mathrm{par}^{(j)}(u_i) = \mathrm{par}(\mathrm{par}^{(j-1)}(u_i))$*. In line 9, we use the following way to find* v_i'*. We initialize* v_i' *as* u_i'*. For* $k = t \to 0$*, if* $\mathrm{dep}_{\mathrm{par}}(g_k(v_i')) > \mathrm{dep}_{\mathrm{par}}(v_i)$ *(i.e.,* $\mathrm{dep}_{\mathrm{par}}(\mathrm{par}'^{(2^k)}(v_i')) > \mathrm{dep}_{\mathrm{par}}(v_i)$*), we set* $v_i' \leftarrow g_k(v_i') = \mathrm{par}'^{(2^k)}(v_i')$.

Lemma 5.2.17 (Generation of multiple paths)**.** *Let* par : $V \to V$ *be a set of parent pointers on a vertex set* V*.* par *has a unique root. Let* $Q = \{(u_1, v_1), (u_2, v_2), \cdots, (u_q, v_q)\} \subseteq V \times V$ *be a set of pairs of vertices where* $\forall j \in [q]$*,* v_j *is an ancestor of* u_j *in* par*. Let* $P_1, P_2, \cdots, P_q$ *be the output of* MULTIPATH(par, Q) *(Algorithm 16). Then* $\forall j \in [q], P_j = P(u_j, v_j)$*, i.e.,* P_j *is a sequence which denotes a path from* u_j *to* v_j *in* par*.*

Proof. Consider a pair $(u_i, v_i) \in Q$. If $\mathrm{dep}_{\mathrm{par}}(u_i) - \mathrm{dep}_{\mathrm{par}}(v_i) \leq 2t$, then P_i will be the path from u_i to v_i in par by line 8.

Algorithm 16 Multi-Paths Generation (space efficient)

1: **procedure** MULTIPATH(par : $V \to V, Q$) ▷ par : $V \to V$ is a set of parent pointers of a rooted tree on a set V of n vertices (par has a unique root r), and Q is a set of q vertex-ancestor pairs $Q = \{(u_1, v_1), (u_2, v_2), \cdots, (u_q, v_q)\}$ where $\forall i \in [q], v_i$ is an ancestor of u_i.
2: $(V', \text{par}') \leftarrow$COMPRESS(par). ▷ Algorithm 12.
3: Compute the depth of each vertex in par. ▷ Run TREECONTRACTION($(V, \emptyset)$, par) (see Algorithm 3). Let the output $g^{(r)} : V \to \mathbb{Z}_{\geq 0}$ be $\text{dep}_{\text{par}} : V \to \mathbb{Z}_{\geq 0}$ (see Lemma 4.2.9).
4: Compute the depth of each vertex in par′. ▷ Run TREECONTRACTION($(V', \emptyset)$, par′) (see Algorithm 3). Let the output $g^{(r)} : V' \to \mathbb{Z}_{\geq 0}$ be $\text{dep}_{\text{par}'} : V \to \mathbb{Z}_{\geq 0}$ (see Lemma 4.2.9).
5: Set $d \leftarrow \text{dep}(\text{par}), t \leftarrow \lceil \log d \rceil$.
6: Compute mappings $g_0, g_1, \cdots g_t : V' \to V'$ such that $\forall v \in V', j \in \{0, 1, \cdots, t\}$, $g_j(v) = \text{par}'^{(2^j)}(v)$. ▷ Run FINDANCESTORS(par) (see Algorithm 6).
7: **for** vertex-ancestor pair $(u_i, v_i) \in Q$ **do**
8: If $\text{dep}_{\text{par}}(u_i) - \text{dep}_{\text{par}}(v_i) \leq 2t$, generate the path sequence $P_i = (u_i, \text{par}^{(1)}(u_i), \text{par}^{(2)}(u_i), \cdots, v_i)$ directly.
9: Otherwise, find the minimum $j \in [2t]$ such that $\text{par}^{(j)}(u_i) \in V'$. Set $u_i' \leftarrow \text{par}^{(j)}(u_i)$. Find an ancestor v_i' of u_i' in par′ such that $\text{dep}_{\text{par}}(v_i') \geq \text{dep}_{\text{par}}(v_i)$ and $\text{dep}_{\text{par}}(v_i') - 2t \leq \text{dep}_{\text{par}}(v_i)$.
10: Generate the path $P'(u_i', v_i')$ in par′. ▷ Run MULTIPATHLARGE(par′, Q'), where Q' contains all (u_i', v_i') (see Algorithm 15).
11: Initialize a sequence A as the concatenation of (u_i), $P'(u_i', v_i')$ and (v_i).
12: Repeat: for each element a_i in A, if a_i is not the last element and $a_{i+1} \neq \text{par}(a_i)$, insert $\text{par}(a_i)$ between a_i and a_{i+1}; until A does not change. Output the final sequence A as the path sequence P_i.
13: **end for**
14: **end procedure**

We only need to consider the case when $\text{dep}_{\text{par}}(u_i) > \text{dep}_{\text{par}}(v_i) + 2t$. According to Lemma 5.2.7, $\exists j \in [2t]$ such that $\text{par}^{(j)}(u_i) \in V'$. Thus, $u_i' \in V'$ can be found by line 9. Then v_i' can be found. v_i is an ancestor of v_i'. v_i' is an ancestor of u_i'. u_i' is an ancestor of u_i. In line 10, the path $P'(u_i', v_i')$ can be found according to Lemma 5.2.14. In line 11, the initialization of A should be $(u_i, u_i', \text{par}'^{(1)}(u_i'), \text{par}'^{(2)}(u_i'), \cdots, v_i', v_i)$. By Lemma 5.2.7, the initialization of A is also $(u_i, u_i', \text{par}^{(t)}(u_i'), \text{par}^{(2t)}(u_i'), \cdots, v_i', v_i)$. Then by line 12, the final sequence $P_i = A$ will be $(u_i, \text{par}^{(1)}(u_i), \text{par}^{(2)}(u_i), \cdots, v_i)$ which denotes the path from u_i to v_i in par. □

5.2.4 Leaf sampling

Given a set of rooted trees, our goal is to sample a set of leaves for each tree, and to give an order of those sampled leaves. The algorithm is shown in Algorithm 17.

Lemma 5.2.18. *Let* par : $V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex*

Algorithm 17 Leaf Sampling

1: **procedure** LEAFSAMPLING(par : $V \to V, m, \delta$) ▷ Lemma 5.2.18
2: Output: $A = (a_1, a_2, \cdots, a_s)$.
3: Let $t \leftarrow \lceil m^{1/3} \rceil$.
4: Compute $L \leftarrow$ leaves(par).
5: Compute rank : $V \to \mathbb{Z}_{\geq 0}$ such that $\forall v \in V, \text{rank}(v) \leftarrow \text{rank}_{\text{par}}(v)$. ▷ Definition 5.2.1
6: If $|V| \leq m$, let $\{a_1, a_2, \cdots, a_s\} \leftarrow L$, and return $A \leftarrow (a_1, a_2, \cdots, a_s)$ which satisfies $a_1 <_{\text{par}} a_2 <_{\text{par}} \cdots <_{\text{par}} a_s$. ▷ $<_{\text{par}}$ follows Definition 5.2.6
7: If $|L| \leq 8t$, let $S \leftarrow L$.
8: Let $p \leftarrow \min(1, 640(1 + \log(m)/\delta)t/|L|)$.
9: If $|L| > t$, sample each $v \in L$ with probability p independently. let S be the set of samples.
10: Compute par′ : $V \to V$ such that $\forall v \in V$, if $\text{child}_{\text{par}}(v) \neq \emptyset$, then $\text{par}'(v) = \text{child}_{\text{par}}(v, 1)$; Otherwise let $\text{par}'(v) = v$. ▷ $\text{par}'(v)$ points to v's first child in par.
11: $(\text{par}'^{(\infty)} : V \to V, G') \leftarrow$ TREECONTRACTION$((V, \emptyset), \text{par}')$. ▷ Algorithm 3.
12: Find $w \in V$ with $\text{par}(w) = w$. ▷ Find the root.
13: Let $a_1 \leftarrow \text{par}'^{(\infty)}(w), S \leftarrow S \cup \{a_1\}$. ▷ Find the first leaf.
14: Let $Q \leftarrow \{(u, v) \mid (u, v) \in S \times S, u \neq v\}$.
15: Let lca $\leftarrow$ LCA(par, Q). ▷ Algorithm 14.
16: Let $s \leftarrow |S|$.
17: **for** $i = 2 \to s$ **do** ▷ Determine the order of sampled leaves.
18: For all $x, y \in S \setminus \{a_1, a_2, \cdots, a_{i-1}\}$, let $(p_{x,y}, p_{xy,x}, p_{xy,y}) = \text{lca}(x, y)$.
19: Find $x^* \in S \setminus \{a_1, a_2, \cdots, a_{i-1}\}$ s.t. $\forall y \in S \setminus \{a_1, a_2, \cdots, a_{i-1}, x^*\}, \text{rank}(p_{x^*y,x^*}) < \text{rank}(p_{x^*y,y})$.
20: Let $a_i \leftarrow x^*$.
21: **end for**
22: **return** $A = (a_1, a_2, \cdots, a_s)$.
23: **end procedure**

set V, and par *has a unique root. Let $m > 0, \delta \in (0, 1)$ be parameters, and let $|V| \leq m^{1/\delta}$. Let $(a_1, a_2, \cdots, a_s) =$ LEAFSAMPLING(par, m, δ) (Algorithm 17). Then it has following properties:*

1. $a_1 <_{\text{par}} a_2 <_{\text{par}} \cdots <_{\text{par}} a_s$.

2. *If $|V| \leq m$ or $|\text{leaves(par)}| \leq 8\lceil m^{1/3} \rceil$, then $\{a_1, a_2, \cdots, a_s\} = \text{leaves(par)}$. Otherwise, with probability at least $1 - 1/(100m^{5/\delta})$, $\forall v \in \text{leaves(par)} \setminus \{a_1\}$, there is a vertex $w \in \{a_1, a_2, \cdots, a_s\}$ such that $w <_{\text{par}} v$ and the number of leaves between w and v is at most $|\text{leaves(par)}|/\lceil m^{1/3} \rceil$, i.e. $|\{u \in \text{leaves(par)} \mid w <_{\text{par}} u <_{\text{par}} v\}| \leq |\text{leaves(par)}|/\lceil m^{1/3} \rceil$.*

3. *If $|V| > m$ and $|\text{leaves(par)}| > 8\lceil m^{1/3} \rceil$, then with probability at least $1 - 1/(100m^{5/\delta})$, $s = |S| = |\{a_1, a_2, \cdots, a_s\}| \leq 960\lceil m^{1/3} \rceil (1 + \log(m)/\delta)$.*

Proof. Firstly, let us focus on property 1. According to line 11 to line 13 and Lemma 4.3.6, we

know $\forall k \in \mathbb{Z}_{\geq 0}\, \mathrm{rank}_{\mathrm{par}}(\mathrm{par}^{(k)}(a_1)) = 1$, and $\mathrm{par}'(a_1) = a_1$ which implies that a_1 is a leaf. Due to Definition 5.2.4, we know that a_1 must be the first leaf appeared in the DFS sequence of par. We can prove the property by induction. Suppose we already have $a_1 <_{\mathrm{par}} a_2 <_{\mathrm{par}} \cdots <_{\mathrm{par}} a_{i-1}$. According to line 18 and Lemma 5.2.12, p_{a_{i-1},a_i} is the LCA of (a_{i-1}, a_i). $p_{a_{i-1}a_i,a_{i-1}}$ is a child of p_{a_{i-1},a_i} and is an ancestor of a_{i-1}. $p_{a_{i-1}a_i,a_i}$ is a child of p_{a_{i-1},a_i} and is an ancestor of a_i. By Fact 5.2.5, since $\mathrm{rank}(p_{a_{i-1}a_i,a_{i-1}}) < \mathrm{rank}(p_{a_{i-1}a_i,a_i})$, we have $a_{i-1} <_{\mathrm{par}} a_i$. To conclude, we have $a_1 <_{\mathrm{par}} a_2 <_{\mathrm{par}} \cdots <_{\mathrm{par}} a_s$.

For property 2, if $|V| \leq m$ or $|\,\mathrm{leaves}(\mathrm{par})| \leq 8\lceil m^{1/3}\rceil$, then by line 6 and line 7, we know $\{a_1, a_2, \cdots, a_s\} = \mathrm{leaves}(\mathrm{par})$.

Now consider the case when $|V| > m$ and $|\,\mathrm{leaves}(\mathrm{par})| > 8\lceil m^{1/3}\rceil$. Let $t = \lceil m^{1/3}\rceil$. Let $\mathrm{leaves}(\mathrm{par}) = \{u_1, u_2, \cdots, u_q\}$, and let $u_1 <_{\mathrm{par}} u_2 <_{\mathrm{par}} \cdots <_{\mathrm{par}} u_q$. Let us partition $u_1, \cdots, u_q$ into $4 \cdot t$ groups $G_1 = \{u_1, u_2, \cdots, u_{\lfloor q/(4t)\rfloor}\}, G_2 = \{u_{\lfloor q/(4t)\rfloor+1}, u_{\lfloor q/(4t)\rfloor+2}, \cdots, u_{2\cdot\lfloor q/(4t)\rfloor}\}, \cdots, G_{4t} = \{u_{(4t-1)\lfloor q/(4t)\rfloor+1}, u_{(4t-1)\lfloor q/(4t)\rfloor+2}, \cdots, u_q\}$. Then each group has size at least $q/(8t)$ and at most $q/(2t)$. For a certain G_i, by Chernoff bound, we have

$$\begin{aligned}
&\Pr\left(|G_i \cap S| \leq \frac{1}{2}\cdot\frac{q}{8t}\cdot p\right)\\
\leq\ &\exp\left(-\frac{1}{8}\cdot\frac{q}{8t}\cdot p\right)\\
\leq\ &1/(100m^{10/\delta})
\end{aligned}$$

where the last inequality follows by $p = \min(1, (10 + 10\log(m)/\delta)\cdot 64t/q)$. Notice that $q \leq |V| \leq m^{1/\delta}$. We can take union bound over all G_i. Then with probability at least $1 - 1/(100m^{5/\delta})$, $\forall i \in [4t], G_i \cap S \neq \emptyset$. Thus, $\forall v \in \mathrm{leaves}(\mathrm{par})$, there is a vertex $w \in \{a_1, a_2, \cdots, a_s\}$ such that $w <_{\mathrm{par}} v$ and the number of leaves between w and v is at most $|\,\mathrm{leaves}(\mathrm{par})|/\lceil m^{1/3}\rceil$, i.e. $|\{u \in \mathrm{leaves}(\mathrm{par}) \mid w <_{\mathrm{par}} u <_{\mathrm{par}} v\}| \leq |\,\mathrm{leaves}(\mathrm{par})|/\lceil m^{1/3}\rceil$.

For property 3, by applying Chernoff bound, we have

$$\Pr\left(|S \cap \mathrm{leaves}(\mathrm{par})| \geq \frac{3}{2}|\,\mathrm{leaves}(\mathrm{par})|\cdot p\right)$$

$$\leq \exp\left(-\frac{1}{12} \cdot |\operatorname{leaves}(\mathrm{par})| \cdot p\right)$$
$$\leq 1/(100m^{10/\delta})$$

where the last inequality follows by $p = \min(1, (10 + 10\log(m)/\delta) \cdot 64t/|\operatorname{leaves}(\mathrm{par})|)$.

Since $\frac{3}{2}|\operatorname{leaves}(\mathrm{par})| \cdot p \leq 960\lceil m^{1/3}\rceil(1 + \log(m)/\delta)$, we complete the proof. □

5.2.5 DFS subsequence

Let par : $V \to V$ be a set of parent pointers on a vertex set V, and par has a unique root v. Let $\{u_1, u_2, \cdots, u_q\} = \operatorname{leaves}(\mathrm{par})$, and $u_1 <_{\mathrm{par}} u_2 <_{\mathrm{par}} \cdots <_{\mathrm{par}} u_q$. One observation is that the DFS sequence of par can be generated in the following way:

1. The first part of the DFS sequence is the path from v to u_1.
2. Then it follows by the path from $\mathrm{par}(u_1)$ to the LCA of (u_1, u_2), the path from one of the child of the LCA of (u_1, u_2) to u_2, the path from $\mathrm{par}(u_2)$ to the LCA of (u_2, u_3), the path from one of the child of the LCA of (u_2, u_3) to $u_3, \cdots$, the path from one of the child of the LCA of (u_{q-1}, u_q) to u_q.
3. The last part of the DFS sequence is a path from $\mathrm{par}(u_q)$ to v.

Fact 5.2.19. *Let* par : $V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V*, and* par *has a unique root* v*. Let* $\{u_1, u_2, \cdots, u_q\} =$ leaves(par) *(See Definition 5.2.2), and* $u_1 <_{\mathrm{par}} u_2 <_{\mathrm{par}} \cdots <_{\mathrm{par}} u_q$*. Let* $A = (a_1, a_2, \cdots, a_m)$ *be the DFS sequence (See Definition 5.2.4) of* par*. Then,*

1. *If* u_1 *appears at* a_i*, then* $(a_1, a_2, \cdots, a_i)$ *is the path from* v *to* u_1*.*
2. $\forall i \in [q-1]$*, if* u_i *appears at* a_j*, and* u_{i+1} *appears at* a_k*, then* $\exists j < t < k$ *such that* a_t *is the LCA of* (u_i, u_{i+1})*. In addition,* $(a_j, a_{j+1}, \cdots, a_t)$ *is the path from* a_j *to* a_t*, and* $(a_t, a_{t+1}, \cdots, a_k)$ *is the path from* a_t *to* a_k*.*

Algorithm 18 DFS Subsequence

procedure SUBDFS(par : $V \to V, m, \delta$) ▷ Lemma 5.2.22, Lemma 5.2.23.
  Output: $V' \subseteq V, A = (a_1, a_2, \cdots, a_s)$.
  If $V = \{v\}$, return $V' \leftarrow V, A \leftarrow (v)$.
  Let v be the root in par, i.e. $\text{par}(v) = v$.
  $L = (l_1, l_2, \cdots, l_t) \leftarrow$ LEAFSAMPLING(par, m, δ). ▷ Algorithm 17.
  $Q = \{(l_i, l_{i+1}) \mid i \in [t-1]\}$.
  lca = LCA(par, Q). ▷ Algorithm 14.
  $\forall i \in [t-1], (p_{l_i,l_{i+1}}, p_{i,l_i}, p_{i,l_{i+1}}) = \text{lca}(l_i, l_{i+1})$.
  $Q' = \{(l_1, v), (\text{par}(l_1), p_{l_1,l_2}), (l_2, p_{1,l_2}), (\text{par}(l_2), p_{l_2,l_3}), (l_3, p_{2,l_3}), \cdots, (l_t, p_{t-1,l_t}), (\text{par}(l_t), v)\}$.
  $\{\text{path sequence } P_i \mid i \in [2t]\} \leftarrow$ MULTIPATH(par, Q'). ▷ Algorithm 16.
  $V' \leftarrow \bigcup_{i=1}^{2t} \{v \in V \mid v \text{ appears in } P_i\}$.
  Let $\text{par}' : V' \to V'$ satisfy $\forall v \in V', \text{par}'(v) = \text{par}(v)$.
  for $i \in \{1, 3, 5, \cdots, 2t-1\}$ **do**
    Compute $A'_i \leftarrow P_i$.
  end for
  for $i \in \{2, 4, 6, \cdots, 2t\}$ **do**
    Compute A'_i as the inverse of P'_i, i.e., $A'_i \leftarrow (u_{|P_i|}, u_{|P_i|-1}, \cdots, u_1)$ for $P_i = (u_1, u_2, \cdots, u_{|P_i|})$.
  end for
  Let $A' \leftarrow A'_1 A'_2 \cdots A'_{2t}$. ▷ A' is the concatenation of $A'_1, A'_2, \cdots, A'_{2t}$.
  $\forall u \in V'$, compute $\text{rank}_{\text{par}}(u)$ and $\text{rank}_{\text{par}'}(u)$.
  $\forall u \in V', i \in [|\text{child}_{\text{par}'}| + 1]$ compute $\text{pos}(u, i) = j$ such that the j^{th} element in A' is the i^{th} time that u appears.
  Let b be the length of A'.
  Initialize $c : [b] \to \mathbb{Z}_{\geq 0}$. ▷ c determine the number of copies needed for each element in A'.
  for $u \in V' \setminus \{v\}$ **do**
    If $u \in \text{leaves}(\text{par}')$, let $c(\text{pos}(u, 1)) \leftarrow 1$. ▷ A leaf should only have one copy.
    If $\text{rank}_{\text{par}'}(u) = 1$, let $c(\text{pos}(\text{par}'(u), 1)) \leftarrow \text{rank}_{\text{par}}(u)$.
    If $\text{rank}_{\text{par}'}(u) = |\text{child}_{\text{par}'}(\text{par}'(u))|$, let $c(\text{pos}(\text{par}'(u), \text{rank}_{\text{par}'}(u) + 1)) \leftarrow |\text{child}_{\text{par}}(\text{par}(u))| + 1 - \text{rank}_{\text{par}}(u)$.
    If $1 \leq \text{rank}_{\text{par}'}(u) < |\text{child}_{\text{par}'}(\text{par}'(u))|$, let $c(\text{pos}(\text{par}'(u), \text{rank}_{\text{par}'}(u) + 1)) \leftarrow \text{rank}_{\text{par}}(\text{child}_{\text{par}'}(\text{par}'(u), \text{rank}_{\text{par}'}(u) + 1)) - \text{rank}_{\text{par}}(u)$.
  end for
  For each $j \in [b]$, duplicate the j^{th} element of A' $c(j)$ times. Let A be the obtained sequence.
  return V', A.
end procedure

3. *If u_q appears at a_i, then $(a_i, a_{i+1}, \cdots, a_m)$ is the path from u_q to v.*

Proof. Property 1, 3 follows by the definition of DFS sequence (See Definition 5.2.4) and a simple induction.

Now consider property 2. Since A is a DFS sequence, $\forall l \in \{j, j+1, \cdots, k-1\}$, either $\mathrm{par}(a_l) = a_{l+1}$ or $\mathrm{par}(a_{l+1}) = a_l$. Thus, the path between u_i and u_{i+1} is a subsequence of $(a_j, a_{j+1}, \cdots, a_k)$. If $\mathrm{par}(a_{l+1}) = a_l$ but a_{l+1} is not on the path between u_i and u_{i+1}, then there must be a leaf x in the subtree of a_{l+1} which implies $u_i <_{\mathrm{par}} x <_{\mathrm{par}} u_{i+1}$, and thus leads to a contradiction. If $\mathrm{par}(a_l) = a_{l+1}$ but a_{l+1} is not on the path between u_i and u_{i+1}, then both u_i and u_{i+1} should be in the subtree of a_l, and both of u_i and u_{i+1} should be in the DFS sequence of the subtree of a_l. But we know a_{l+1} cannot be in the DFS sequence of the subtree of a_l. This leads to a contradiction. □

Lemma 5.2.20. *Let* $\mathrm{par} : V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V*, with a unique root. Let* $v \in V$*. Let* $V' = V \setminus \{u \in V \mid v \text{ is an ancestor (See Definition 5.2.8) of } u\}$*, i.e.,* V' *denotes the vertices outside the subtree of* v*. Let* $\mathrm{par}' : V' \to V'$ *satisfy* $\forall u \in V', \mathrm{par}'(u) = \mathrm{par}(u)$*. Then the DFS sequence (See Definition 5.2.4) of* par' *is a subsequence of the DFS sequence of* par*.*

Proof. The proof follows by the property 3 of Fact 5.2.5 directly. □

Corollary 5.2.21. *Let* $\mathrm{par} : V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V*, and* par *has a unique root. Let* $v_1, v_2, \cdots, v_t$ *be* t *vertices in* V*. Let* $V' = V \setminus \{u \in V \mid \exists v \in \{v_1, \cdots, v_t\}, v \text{ is an ancestor (See Definition 5.2.8) of } u\}$*. Let* $\mathrm{par}' : V' \to V'$ *satisfy* $\forall u \in V', \mathrm{par}'(u) = \mathrm{par}(u)$*. Then the DFS sequence (See Definition 5.2.4) of* par' *is a subsequence of the DFS sequence of* par*.*

Proof. The proof is by induction on t. When $t = 1$, then the statement is true by Lemma 5.2.20. Suppose the statement is true for $t-1$. Let $V'' = V \setminus \{u \in V \mid \exists v \in \{v_1, \cdots, v_{t-1}\}, v \text{ is an ancestor of } u\}$, and let $\mathrm{par}'' : V'' \to V''$ satisfy $\forall v \in V'', \mathrm{par}''(v) = \mathrm{par}(v)$. By induction hypothesis, the DFS sequence of par'' is a subsequence of the DFS sequence of par. If one of the $v_1, \cdots, v_{t-1}$ is an ancestor of v_t, then $\mathrm{par}' = \mathrm{par}''$, thus, the DFS sequence of par' is a subsequence of the DFS sequence of par. Otherwise, we have $V' = V'' \setminus \{u \in V'' \mid v_t \text{ is an ancestor of } u\}$. By Lemma 5.2.20, the DFS sequence of par' is a subsequence of the DFS sequence of par'' . Thus, the DFS sequence of par' is a subsequence of the DFS sequence of par. □

Lemma 5.2.22 (Removing several subtrees)**.** *Let* par : $V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V*, and* par *has a unique root. Let* $m > 0, \delta \in (0,1)$ *be parameters, and let* $|V| \leq m^{1/\delta}$*. Let* $(V', A) = \textsc{SubDFS}(\text{par}, m, \delta)$ *(Algorithm 18). Then* $\forall u \in V'$*, we have* $\text{par}(u) \in V'$*. Furthermore, with probability at least* $1 - 1/(100m^{5/\delta})$*,* $\forall u \in V \setminus V'$*, the number of leaves (See Definition 5.2.2) in the subtree (See Definition 5.2.3) of* u *is at most* $\lfloor |\text{leaves(par)}|/\lceil m^{1/3} \rceil \rfloor$*.*

Proof. By Lemma 5.2.18, we know $L \subseteq \text{leaves(par)}$, and $l_1 <_{\text{par}} l_2 <_{\text{par}} \cdots <_{\text{par}} l_t$.

We first prove $\forall u \in V', \text{par}(u) \in V'$. Our proof is by induction on the leaf l_i. By Lemma 5.2.12, we have that $\forall i \in [t-1], p_{l_i,l_{i+1}}$ is the LCA of (l_i, l_{i+1}), $p_{i,l_{i+1}}$ is an ancestor of l_{i+1}, and $p_{i,l_{i+1}} \neq p_{l_i,l_{i+1}}, \text{par}(p_{i,l_{i+1}}) = p_{l_i,l_{i+1}}$. By Lemma 5.2.17, P_1 is the path from l_1 to the root v. P_1 contains all the ancestors of l_1. Thus, every ancestor u of l_1 appears in P_1 and satisfies $\text{par}(u) \in V'$. P_2 is the path from l_1 to an ancestor of l_1. Thus, P_2 is a subsequence of P_1. Suppose now the set of vertices appeared in at least one of paths $P_1, P_2, \cdots, P_{2i-2}$ is $\{u \in V \mid \exists j \in [i-1], u \text{ is an ancestor of } l_j\}$. Notice that every vertex on the path from l_i to the ancestor p_{i-1,l_i} appears in P_{2i-1}. Since $\text{par}(p_{i-1,l_i}) = p_{l_{i-1},l_i}$ is also an ancestor of l_{i-1}, we have that the set of vertices appeared in at least one of paths $P_1, P_2, \cdots, P_{2i-2}, P_{2i-1}$ is $\{u \in V \mid \exists j \in [i], u \text{ is an ancestor of } l_j\}$. Since P_{2i} contains all the vertices on the path from l_i to an ancestor of l_i, we have that the set of vertices appeared in at least one of paths $P_1, P_2, \cdots, P_{2i-1}, P_{2i}$ is $\{u \in V \mid \exists j \in [i], u \text{ is an ancestor of } l_j\}$. Therefore, we have $V' = \{u \in V \mid \exists j \in [t], u \text{ is an ancestor of } l_j\}$. Thus, $\forall u \in V'$, we have $\text{par}(u) \in V'$.

By Lemma 5.2.18, with probability at least $1 - 1/(100m^{5/\delta})$, $\forall u \in \text{leaves(par)} \setminus L$, there exists $w \in L, w <_{\text{par}} u$ such that $|\{x \in \text{leaves(par)} \mid w <_{\text{par}} x <_{\text{par}} u\}| \leq \lfloor |\text{leaves(par)}|/\lceil m^{1/3} \rceil \rfloor$. In the following, we condition on that the above event happens. Let $u \in V \setminus V'$. Due to Fact 5.2.5, the DFS sequence of the subtree of u in par must be a consecutive subsequence of the DFS sequence of par . Thus, $\exists x, y \in \text{leaves(par)}$, the leaves in the subtree of u in par is the set $\{z \in \text{leaves(par)} \mid x <_{\text{par}} z <_{\text{par}} y\} \cup \{x\} \cup \{y\}$. If the number of leaves in the subtree of u is more than $\lfloor |\text{leaves(par)}|/\lceil m^{1/3} \rceil \rfloor$, then $\exists l_i \in L$, u is an ancestor of leaf l_i. But $l_i \in V'$ contradicts to $u \notin V'$. Thus, the number of leaves in the subtree of u is at most $\lfloor |\text{leaves(par)}|/\lceil m^{1/3} \rceil \rfloor$. □

Lemma 5.2.23 (A is a subsequence)**.** *Let* par $: V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V*, and* par *has a unique root. Let* $m > 0, \delta \in (0,1)$ *be parameters, and let* $|V| \le m^{1/\delta}$*. Let* $(V', A) = \textsc{SubDFS}(\text{par}, m, \delta)$ *(Algorithm 18). Then* A *is a subsequence of the DFS sequence of* par *. Furthermore,* $\forall u \in V'$*,* u *appears in* A *exactly* $|\text{child}_{\text{par}}(u) + 1|$ *times, and* $\forall u \notin V'$*,* u *does not appear in* A*.*

Proof. We first show that A' is the DFS sequence of par′ .

Claim 5.2.24. A' *is the DFS sequence of* par′ $: V' \to V'$*.*

Proof. By Lemma 5.2.18, we know $\{l_1, l_2, \cdots, l_t\} = L \subseteq \text{leaves}(\text{par})$, and $l_1 <_{\text{par}} l_2 <_{\text{par}} \cdots <_{\text{par}} l_t$. By Lemma 5.2.12, we have that $\forall i \in [t-1], p_{l_i,l_{i+1}}$ is the LCA of (l_i, l_{i+1}), $p_{i,l_{i+1}}$ is an ancestor of l_{i+1}, and $p_{i,l_{i+1}} \ne p_{l_i,l_{i+1}}, \text{par}(p_{i,l_{i+1}}) = p_{l_i,l_{i+1}}$. By Lemma 5.2.17, $\forall i \in [t]$, P_{2i-1} and P_{2i} only contains some ancestors of l_i. Thus, $\text{leaves}(\text{par}') = L$.

According to Lemma 5.2.22 and Corollary 5.2.21, the DFS sequence of par′ is a subsequence of the DFS sequence of par . Thus, we still have $l_1 <_{\text{par}'} l_2 <_{\text{par}'} <_{\text{par}'} \cdots <_{\text{par}'} l_t$. Due to Lemma 5.2.17, P_1 denotes the path from l_1 to the root v, P_{2t} denotes the path from l_t to the root v, $\forall i \in [t-1], P_{2i}$ denotes the path from $\text{par}'(l_i)$ to the LCA of (l_i, l_{i+1}), and P_{2i+1} denotes the path from l_{i+1} to $p_{i,l_{i+1}}$. Thus, A'_1 is the path from the root v to leaf l_1, A'_{2t} is the path from l_t to the root v, $\forall i \in [t-1]$, $A'_{2i}A'_{2i+1}$ is the path from $\text{par}'(l_i)$ to l_{i+1}. Due to Fact 5.2.19, $A' = A'_1 A'_2 \cdots A'_{2t}$ is the DFS sequence of par′ . □

Let us define some notations. Let $\widetilde{A} = (\widetilde{a}_1, \widetilde{a}_2, \cdots, \widetilde{a}_{\widetilde{s}})$ be the DFS sequence of par . $\forall u \in V$, let $\text{st}_{\widetilde{A}}(u) = j$ such that $\widetilde{a}_j$ is the first time that u appears in $\widetilde{A}$. We define $\text{ed}_{\widetilde{A}}(u)$ be the position such that $\widetilde{a}_{\text{ed}_{\widetilde{A}}(u)}$ is the last time that u appears in $\widetilde{A}$. Similarly, $\forall u \in V'$, we can define $\text{st}_{A'}(u), \text{st}_A(u), \text{ed}_{A'}(u), \text{ed}_A(u)$ to be the positions of the first time u appears in A', the first time u appears in A, the last time u appears in A', and the last time u appears in A respectively.

Since v is the root (in both par and par′), it suffices to prove that $(a_{\text{st}_A(v)}, a_{\text{st}_A(v)+1}, \cdots, a_{\text{ed}_A(v)})$ is a subsequence of $(\widetilde{a}_{\text{st}_{\widetilde{A}}(v)}, \widetilde{a}_{\text{st}_{\widetilde{A}}(v)+1}, \cdots, \widetilde{a}_{\text{ed}_{\widetilde{A}}(v)})$. Our proof is by induction on $\text{dep}_{\text{par}}(u)$ for $u \in V'$. If $\text{dep}_{\text{par}}(u) = \text{dep}(\text{par})$, then u must be a leaf in par′ (or par, since par′ and par are the same on V').

In this case, $\mathrm{st}_A(u) = \mathrm{ed}_A(u), \mathrm{st}_{\widetilde{A}}(u) = \mathrm{ed}_{\widetilde{A}}(u)$, and $(a_{\mathrm{st}_A(u)}) = (\widetilde{a}_{\mathrm{st}_{\widetilde{A}}(u)}) = (u)$. Suppose for all $u \in V'$ with $\mathrm{dep}_{\mathrm{par}}(u) > d$, we have that $(a_{\mathrm{st}_A(u)}, \cdots, a_{\mathrm{ed}_A(u)})$ is a subsequence of $(\widetilde{a}_{\mathrm{st}_{\widetilde{A}}(u)}, \cdots, \widetilde{a}_{\mathrm{ed}_{\widetilde{A}}(u)})$. Let u be a vertex in V' with $\mathrm{dep}_{\mathrm{par}}(u) = d$. If u is a leaf, then it is the same as the previous argument. Now let us consider the case when u is not a leaf. According to Claim 5.2.24, A' is the DFS sequence of par′. Due to line 30, A is obtained by duplicating each element of A' several times. Let $w_1, w_2, \cdots, w_k$ be the children of u in par′, and $\mathrm{rank}_{\mathrm{par}'}(w_1) = 1, \mathrm{rank}_{\mathrm{par}'}(w_2) = 2, \cdots, \mathrm{rank}_{\mathrm{par}'}(w_k) = |\mathrm{child}_{\mathrm{par}'}(u)|$. Then, according to Fact 5.2.5, $(a_{\mathrm{st}_A(u)}, \cdots, a_{\mathrm{ed}_A(u)})$ should look like:

$$(u, \cdots, u, a_{\mathrm{st}_A(w_1)}, \cdots, a_{\mathrm{ed}_A(w_1)}, u, \cdots, u, a_{\mathrm{st}_A(w_2)}, \cdots, a_{\mathrm{ed}_A(w_2)}, \cdots, a_{\mathrm{st}_A(w_k)}, \cdots, a_{\mathrm{ed}_A(w_k)}, u, \cdots, u)$$

where the number of u before $a_{\mathrm{st}_A(w_1)}$ is $\mathrm{rank}_{\mathrm{par}}(w_1)$ (see line 26), the number of u before $a_{\mathrm{st}_A(w_i)}$ for $i \in [k] \setminus \{1\}$ is $\mathrm{rank}_{\mathrm{par}}(w_i) - \mathrm{rank}_{\mathrm{par}}(w_{i-1})$ (see line 28), and the number of u after $a_{\mathrm{ed}_A(w_k)}$ is $|\mathrm{child}_{\mathrm{par}}(u)| - \mathrm{rank}_{\mathrm{par}}(w_k) + 1$ (see line 27). Since $\widetilde{A}$ is the DFS sequence of par, according to Fact 5.2.5, the number of u in $\widetilde{A}$ before $\widetilde{a}_{\mathrm{st}_{\widetilde{A}}(w_1)}$ is $\mathrm{rank}_{\mathrm{par}}(w_1)$. By our induction hypothesis, $(a_{\mathrm{st}_A(w_1)}, \cdots, a_{\mathrm{ed}_A(w_1)})$ is a subsequence of $(\widetilde{a}_{\mathrm{st}_{\widetilde{A}}(w_1)}, \cdots, \widetilde{a}_{\mathrm{ed}_{\widetilde{A}}(w_1)})$. Thus, $(a_{\mathrm{st}_A(u)}, \cdots, a_{\mathrm{ed}_A(w_1)})$ is a subsequence of $(\widetilde{a}_{\mathrm{st}_{\widetilde{A}}(u)}, \cdots, \widetilde{a}_{\mathrm{ed}_{\widetilde{A}(w_1)}})$. According to Fact 5.2.5, $\forall i \in [k] \setminus \{1\}$, the number of u in $\widetilde{A}$ between $\widetilde{a}_{\mathrm{ed}_{\widetilde{A}}(w_{i-1})}$ and $\widetilde{a}_{\mathrm{st}_{\widetilde{A}}(w_i)}$ is $\mathrm{rank}_{\mathrm{par}}(w_i) - \mathrm{rank}_{\mathrm{par}}(w_{i-1})$. By our induction hypothesis, for all $i \in [k] \setminus \{1\}$, $(a_{\mathrm{st}_A(w_i)}, \cdots, a_{\mathrm{ed}_A(w_i)})$ is a subsequence of $(\widetilde{a}_{\mathrm{st}_{\widetilde{A}}(w_i)}, \cdots, \widetilde{a}_{\mathrm{ed}_{\widetilde{A}}(w_i)})$. Thus, $(a_{\mathrm{st}_A(u)}, \cdots, a_{\mathrm{ed}_A(w_k)})$ is a subsequence of $(\widetilde{a}_{\mathrm{st}_{\widetilde{A}}(u)}, \cdots, \widetilde{a}_{\mathrm{ed}_{\widetilde{A}(w_k)}})$. According to Fact 5.2.5, the number of u in $\widetilde{A}$ after $\widetilde{a}_{\mathrm{ed}_{\widetilde{A}}(w_k)}$ is $|\mathrm{child}_{\mathrm{par}}(u)| - \mathrm{rank}_{\mathrm{par}}(w_k) + 1$. Thus, $(a_{\mathrm{st}_A(u)}, \cdots, a_{\mathrm{ed}_A(u)})$ is a subsequence of $(\widetilde{a}_{\mathrm{st}_{\widetilde{A}}(u)}, \cdots, \widetilde{a}_{\mathrm{ed}_{\widetilde{A}(u)}})$. Furthermore, the number of u appears in A is $|\mathrm{child}_{\mathrm{par}}(u)| - \mathrm{rank}_{\mathrm{par}}(w_k) + 1 + \mathrm{rank}_{\mathrm{par}}(w_1) + \sum_{i=2}^{k} \mathrm{rank}_{\mathrm{par}}(w_i) - \mathrm{rank}_{\mathrm{par}}(w_{i-1}) = |\mathrm{child}_{\mathrm{par}}(u)| + 1$.

Since A' is the DFS sequence of par′, $\forall u \notin V'$, u does not appear in A'. Thus, $\forall u \notin V'$, u does not appear in A.

□

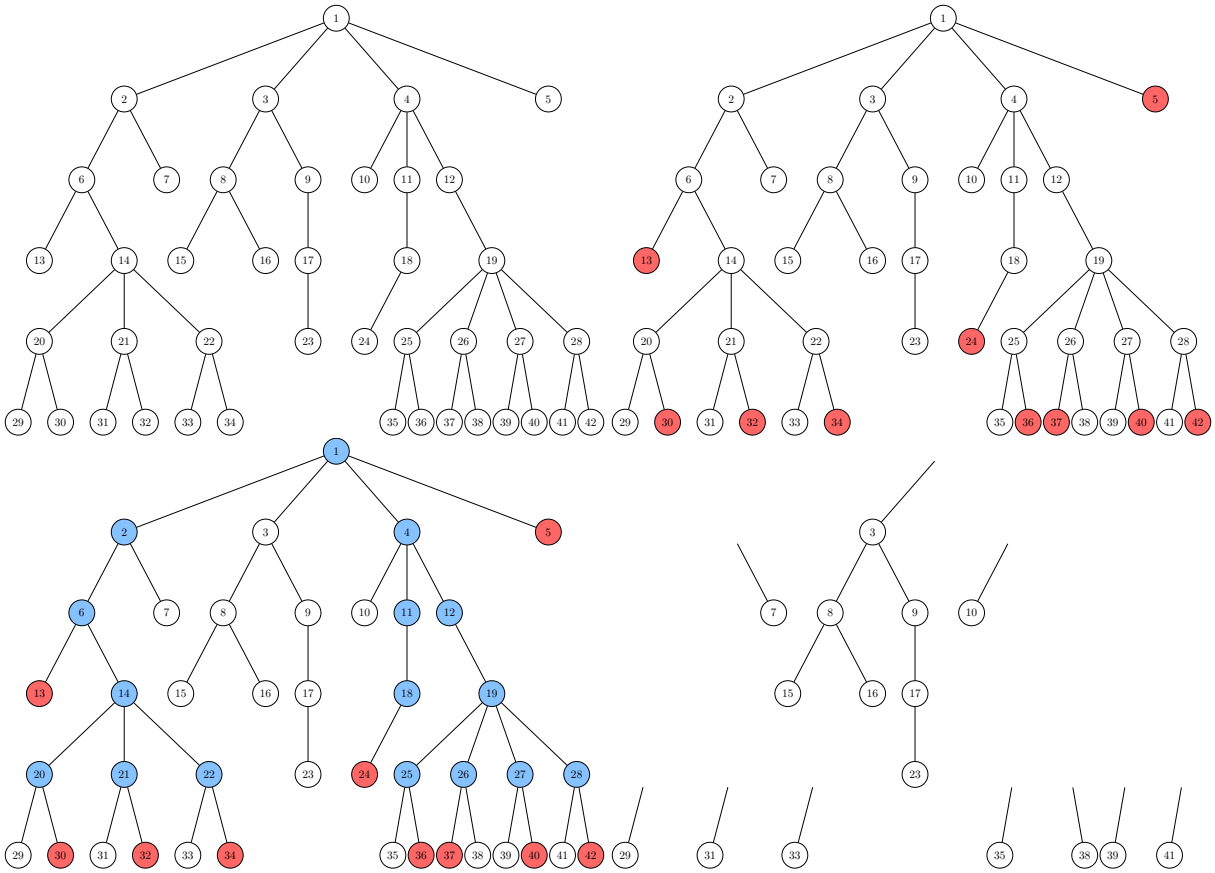

Figure 5.1: Given a tree that has 42 vertices (top-left), we label all the vertices from 1 to 42. Firstly, we sample some leaves (red vertices, i.e. $\{5, 13, 24, 30, 32, 34, 36, 37, 40, 42\}$) in the tree (top-right tree). Then we find a DFS sequence of the tree (the tree formed by all the blue and red vertices in the bottom-left tree) which only contains all the sampled leaves and their ancestors. Finally, we recursively find the DFS sequences of remaining subtrees(bottom-right).

5.2.6 DFS sequence

In this section, we show how to use Algorithm 18 as a subroutine to output a DFS sequence. The high level idea is that we use Algorithm 18 to generate subsequences of the DFS sequence in each iteration, and we ensure that the missing part of the DFS sequence must be the DFS sequences of many subtrees. After the i^{th} iteration, we should ensure that the number of leaves of each subtree which has a missing DFS sequence is at most n/m^i, where m is some parameter depending on some computational resources (e.g. memory size of a machine). The description of the algorithm is shown in Algorithm 19. Figure 5.1 shows one step in our algorithm.

Theorem 5.2.25 (Correctness of DFS sequence)**.** *Let* par $: V \to V$ *be a set of parent pointers*

Algorithm 19 DFS Sequence

procedure DFS(par : $V \to V, m$) ▷ Theorem 5.2.25, Theorem 5.2.27.
  Output: FAIL or $A = (a_1, a_2, \cdots, a_{2|V|-1})$.
  $n \leftarrow |V|, \delta \leftarrow 1/\log_m n$.
  Let $\mathrm{par}_0 = \mathrm{par}$.
  $(V_0, A_0) \leftarrow \textsc{SubDFS}(\mathrm{par}_0, m, \delta)$. ▷ Algorithm 18.
  Let $r \leftarrow \lceil 3/\delta \rceil + 2$.
  for $i = 1 \to r$ **do** ▷ $v \in V_i \Leftrightarrow v$ appears in A_i.
    ▷ If $v \in V_i$, then v appears $|\mathrm{child}_{\mathrm{par}}(v)| + 1$ times in A_i.
    Let $V_i' \leftarrow V \setminus V_{i-1}$.
    Initialize $\mathrm{par}_i : V_i' \to V_i'$.
    For $v \in V_i'$, if $\mathrm{par}(v) \in V_{i-1}$, let $\mathrm{par}_i(v) \leftarrow v$; Otherwise, let $\mathrm{par}_i(v) \leftarrow \mathrm{par}(v)$.
    $(\mathrm{par}_i^{(\infty)}, (V_i'', \emptyset)) \leftarrow \textsc{TreeContraction}((V_i', \emptyset), \mathrm{par}_i)$. ▷ Algorithm 3.
    $V_i \leftarrow V_{i-1}$.
    $A_i \leftarrow A_{i-1}$.
    for $v \in V_i', \mathrm{par}_i(v) = v$ **do** ▷ The DFS sequence of the subtree of v in par is missing.
      Let $V_i'(v) \leftarrow \{u \in V_i' \mid \mathrm{par}_i^{(\infty)}(u) = v\}$.
      Let $\mathrm{par}_{i,v} : V_i'(v) \to V_i'(v)$ satisfy $\forall u \in V_i'(v), \mathrm{par}_{i,v}(u) = \mathrm{par}_i(u)$.
      Let $(V_{i,v}, A_{i,v}) \leftarrow \textsc{SubDFS}(\mathrm{par}_{i,v}, m, \delta)$. ▷ Algorithm 18.
      $V_i \leftarrow V_i \cup V_{i,v}$.
      Insert $A_{i,v}$ after the $\mathrm{rank}_{\mathrm{par}}(v)^{\mathrm{th}}$ time appearance of v in A_i.
    end for
  end for
  If $V_r = V$, return A_r as A. Otherwise, return FAIL.
end procedure

(See Definition 4.2.5) on a vertex set V, and par *has a unique root. Let $n = |V|, m = n^\delta$ for some constant $\delta \in (0, 1)$. If $A = \mathrm{DFS}(\mathrm{par}, m)$ (Algorithm 19) does not output* FAIL*, then A is the DFS sequence of* par.

Proof. It suffices to prove the following claim.

Claim 5.2.26. *Let $i \in \{0\} \cup [r]$. A_i is a subsequence of the DFS sequence of* par. *$\forall v \in V_i$, $\mathrm{par}(v) \in V_i$. Furthermore, $\forall v \in V_i$, v appears in A_i exactly $|\mathrm{child}_{\mathrm{par}}(v)| + 1$ times, and $\forall v \notin V_i$, v does not appear in A_i.*

Proof. Our proof is by induction on i. If $i = 0$, then by Lemma 5.2.23, A_0 is a subsequence of the DFS sequence of par, $\forall v \in V_0$, v appears in A_0 exactly $|\mathrm{child}_{\mathrm{par}}(v)| + 1$ times, and $\forall v \notin V_0$, v does not appear in A_0. By Lemma 5.2.22, we have $\forall v \in V_0, \mathrm{par}(v) \in V_0$.

Suppose the claim is true for $i - 1$. Let $u \in V_i$.

If $u \in V_{i-1}$, then since $V_{i-1} \subseteq V_i$, $\mathrm{par}(u) \in V_i$. Otherwise $u \in V_{i,v}$ for some v with $\mathrm{par}_i(v) = v$. If $u = v$, then $\mathrm{par}(v) \in V_{i-1} \subseteq V_i$. Otherwise, by Lemma 5.2.22, $\mathrm{par}(u) \in V_{i,v} \subseteq V_i$.

Now consider the property of A_i. If $u \in V_{i-1}$, then since A_{i-1} is a subsequence of A_i, and by Lemma 5.2.23 u cannot appear in any $A_{i,v}$, u must appear in A_i exactly $|\mathrm{child}_{\mathrm{par}}(u)| + 1$ times. Otherwise $u \in V_{i,v}$ for some v with $\mathrm{par}_i(v) = v$. By Lemma 5.2.23, u must appear in $A_{i,v}$ $|\mathrm{child}_{\mathrm{par}_{i,v}}(u)| + 1 = |\mathrm{child}_{\mathrm{par}}(u)| + 1$ times. Since u cannot appear in A_{i-1}, u must appear in A_i exactly $|\mathrm{child}_{\mathrm{par}}(u)| + 1$ times. For $v \in V'_i$ with $\mathrm{par}_i(v) = v$, according to Fact 5.2.5 and $\forall w \in \{x \in V \mid v \text{ is an ancestor of } x\}$, w cannot be in V_{i-1}, the $\mathrm{rank}_{\mathrm{par}}(v)^{\mathrm{th}}$ time appearance of v and the $(\mathrm{rank}_{\mathrm{par}}(v) + 1)^{\mathrm{th}}$ time appearance of v should be adjacent in A_{i-1}. Due to Lemma 5.2.23, $A_{i,v}$ is a subsequence of the DFS sequence of the subtree of v in par . Due to Fact 5.2.5, A_i is still a subsequence of the DFS sequence of par after insertion of the sequence $A_{i,v}$.

For any $x \notin V_i$, by Lemma 5.2.23, x cannot be in any $A_{i,v}$. By our induction hypothesis, x cannot be in A_{i-1}. Thus, x cannot be in A_i. □

If the procedure does not output FAIL, then according to the above Claim 5.2.26, $\forall v \in V_r = V$, v appears in $A_r = A$ exactly $|\mathrm{child}_{\mathrm{par}}(v)| + 1$ times, and $A_r = A$ is a subsequence of the DFS sequence of par. Due to Fact 5.2.5, $A = A_r$ is the DFS sequence of par. □

The following lemma claims the success probability of Algorithm 19.

Theorem 5.2.27 (Success probability)**.** *Let* $\mathrm{par} : V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V*, and* par *has a unique root. Let* $n = |V|, m = n^{\delta}$ *for some constant* $\delta \in (0, 1)$*. With probability at least* $1 - 1/(100n^4)$, $A = \mathrm{DFS}(\mathrm{par}, m)$ *(Algorithm 19) does not output* FAIL.

Proof. $\forall i \in [r], v \in V'_i$ with $\mathrm{par}_i(v) = v$, let $\mathcal{E}_{i,v}$ be the event that $\forall u \in V'_i(v) \setminus V_{i,v}$, the number of leaves in the subtree of u in par is at most $|\mathrm{leaves}(\mathrm{par}_{i,v})|/n^{\delta/3}$. Notice that due to Lemma 5.2.22, if $\mathrm{par}_i(v) = v$, then v will be in V_i. Thus, we use $\mathcal{E}_v$ to denote the event $\mathcal{E}_{i,v}$. By Lemma 5.2.22, $\mathcal{E}_v$ happens with probability at least $1 - 1/(100n^5)$. By taking union bound over all v, with probability at least $1 - 1/(100n^4)$, all the events $\mathcal{E}_v$ will happen.

Claim 5.2.28. *Condition on all the events $\mathcal{E}_v$ happen. $\forall i \in [r], v \in V'_i$ with $\mathrm{par}_i(v) = v$, we have $|\operatorname{leaves}(\mathrm{par}_{i,v})| \leq n/n^{(i-1)\delta/3}$.*

Proof. When $i = 1$, the claim is obviously true, since $|\operatorname{leaves}(\mathrm{par}_{i,v})| \leq n$. Suppose the claim holds for $i-1$. Let $v \in V'_i$ with $\mathrm{par}_i(v) = v$. There must be $v' \in V'_{i-1}$ with $\mathrm{par}_{i-1}(v') = v'$, and $v \in V'_{i-1}(v') \setminus V_{i-1,v'}$. Since $\mathcal{E}_{v'}$ happens, the number of leaves in the subtree of v in par is at most $|\operatorname{leaves}(\mathrm{par}_{i-1,v'})|/n^{\delta/3} \leq n/n^{(i-1)\delta/3}$. □

If $V'_r \neq \emptyset$, then $\exists v \in V'_r$ with $\mathrm{par}_i(v) = v$ and $|\operatorname{leaves}(\mathrm{par}_{i,v})| \geq 1$. If all the events $\mathcal{E}_v$ happens, it will contradict to Claim 5.2.28. Thus, if all the events $\mathcal{E}_v$ happens, V'_r must be $\emptyset$, and thus $V_r = V$ which implies that the procedure will not fail. □

5.3 Implementation of DFS sequence in the MPC model

In this section, we show how to implement the subroutines introduced by previous sections in the MPC model. For basic operations in MPC model, we refer readers to Section 2.3.

5.3.1 Compressed rooted tree

Lemma 5.3.1. *Let* par $: V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set V. Suppose* par *has a unique root. Let $n = N = |V|$.* Compress(par) *(Algorithm 12) can be implemented in $(0, \delta)$-MPC model for any constant $\delta \in (0, 1)$ in $O(\log(\mathrm{dep}(\mathrm{par})))$ parallel running time.*

Proof. Consider the implementation of Compress(par $: V \to V$) (Algorithm 12) in the MPC model. The input size is $|V| = n$. In the first step, we need to compute the depth of every vertex in par. This can be computed in the MPC model with $O(n)$ total space and $\Theta(n^\delta)$ local memory size per machine for any constant $\delta \in (0, 1)$ in $O(\log(\mathrm{dep}(\mathrm{par})))$ time (see Algorithm 3, Lemma 4.4.2, and Lemma 4.2.9). Next, we can simultaneously compute $\mathrm{par}^{(t)}(v)$ for every vertex $v \in V$. It takes $O(t) = O(\log(\mathrm{dep}(\mathrm{par})))$ time (each step can be done by **Multiple queries**). In the next step, V' can be computed in $O(1)$ time. Finally $\mathrm{par}'(v)$ for each $v \in V'$ is already computed. It is easy to verify

that the total space needed for each step is $O(n)$. Therefore, COMPRESS(par) can be implemented in the $(0, \delta)$-MPC model for any constant $\delta \in (0, 1)$ in $O(\log(\text{dep}(\text{par})))$ time. □

5.3.2 Lowest common ancestor and multi-paths generation

Lemma 5.3.2. *Let* par : $V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V. *Let* $Q = \{(u_1, v_1), (u_2, v_2), \cdots, (u_q, v_q)\}$ *be a set of* q *pairs of vertices, and* $\forall i \in [q], u_i \neq v_i$, *neither* u_i *nor* v_i *is the LCA of* (u_i, v_i). *Let* $n = |V|, N = n + q$. LCALARGE(par, Q) *(Algorithm 13) can be implemented in* $(\gamma, \delta) -$ MPC *model for any* γ *satisfying* $N^{1+\gamma} \geq q + n \log(\text{dep}(\text{par}))$ *and any constant* $\delta \in (0, 1)$ *in* $O(\log(\text{dep}(\text{par})))$ *parallel running time.*

Proof. By Lemma 4.4.6, line 3 can be implemented in space $O(n \log(\text{dep}(\text{par})))$ and $O(\log(\text{dep}(\text{par})))$ parallel running time. It is easy to see that all the other steps in the procedure can be done by the operations shown in **Multiple queries** and can be done in $O(r) = O(\log(\text{dep}(\text{par})))$ parallel time.

Thus, the total space needed is $O(q+n \log(\text{dep}(\text{par})))$ and the parallel running time is $O(\log(\text{dep}(\text{par})))$.

□

Lemma 5.3.3. *Let* par : $V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V. *Let* $Q = \{(u_1, v_1), (u_2, v_2), \cdots, (u_q, v_q)\}$ *be a set of* q *pairs of vertices, and* $\forall i \in [q], u_i \neq v_i, u_i, v_i \in \text{leaves}(\text{par})$. *Let* $n = |V|, N = n + q$. LCA(par, Q) *(Algorithm 14) can be implemented in* $(0, \delta) -$ MPC *model for any constant* $\delta \in (0, 1)$ *in* $O(\log(\text{dep}(\text{par})))$ *parallel running time.*

Proof. Consider the implementation of LCA(par : $V \to V, Q$) (Algorithm 14) in the MPC model. The input size is $|V| + |Q| = n + q$. The first step computes a compressed rooted tree par′ : $V' \to V'$. According to Lemma 5.3.1, this only requires $O(n)$ total space and $\Theta(n^\delta)$ local memory per machine for any constant $\delta \in (0, 1)$. Before line 6, we need to compute the depth of each vertex in par and the depth of each vertex in par′. According to Algorithm 3, Lemma 4.2.9 and Lemma 4.4.2, it can be done in $(0, \delta)$-MPC model and only take $O(\log(\text{dep}(\text{par})) + \log(\text{dep}(\text{par}'))) = O(\log(\text{dep}(\text{par})))$ time. In line 6, according to Algorithm 6, Lemma 4.3.6 and Lemma 4.4.6, $g_0(\cdot) \equiv \text{par}'^{(2^0)}(\cdot), g_1 \equiv \text{par}'^{(2^1)}(\cdot), \cdots, g_t \equiv \text{par}'^{(2^t)}(\cdot) : V' \to V'$ for $t = \lceil \log(\text{dep}(\text{par})) \rceil$ can be computed in

the MPC model with $O(|V'|\log(\text{dep}(\text{par}')))$ total space and $O(|V'|^{\delta})$ local memory per machine for any constant $\delta \in (0,1)$ in $O(\log(\text{dep}(\text{par}'))) = O(\log(\text{dep}(\text{par})))$ time. According to Lemma 5.2.7, $|V'| \leq |V|/\log(\text{dep}(\text{par}))$. Thus, line 6 only needs $O(n)$ total space and takes time $O(\log(\text{dep}(\text{par})))$. For step 7, we can handle all the queries in Q simultaneously. For line 8, we can use $O(1)$ time to check whether $\text{dep}_{\text{par}}(u_i) > \text{dep}_{\text{par}}(v_i) + 2t$ (see **Multiple queries**). If it is true, we can use $O(t) = O(\log(\text{dep}(\text{par})))$ time to find a $j \in \{0, 1, \cdots, 2t\}$ such that $\text{par}^{(j)}(u_i) \in V'$ (see **Multiple queries**). Then, we apply an exponential search by using $g_0, g_1, \cdots, g_t$ to find $\widehat{u_i}$ (see **Multiple queries**). This takes $O(t) = O(\log(\text{dep}(\text{par})))$ time. Line 9 checks whether $\text{par}^{(j)}(\widehat{u_i})$ is the LCA for every $j \in [4t]$ (see **Multiple queries**). Thus, it takes $O(t) = O(\log(\text{dep}(\text{par})))$ time. In line 10, according to Lemma 5.2.7, there exists $j \in \{0, 1, 2, \cdots, 2t\}$ such that $\text{par}^{(j)}(\widehat{u_i}) \in V'$. Thus, we only need time $O(t)$ to find u_i' (see **Multiple queries**). Similarly, we only need time $O(t)$ to find v_i' (see **Multiple queries**). In step 11, according to Algorithm 13, Lemma 5.3.2 and Lemma 5.2.10, the LCA of each (u_i', v_i') in par′ can be computed simultaneously in the MPC model with $O(|V'|\log(\text{dep}(\text{par}')) + |Q|) = O(|V|/\log(\text{dep}(\text{par})) \cdot \log(\text{dep}(\text{par})) + |Q|) = O(N)$ total space in $O(\log(\text{dep}(\text{par}'))) = O(\log(\text{dep}(\text{par})))$ time. The last step checks (see **Multiple queries**) whether $\text{par}^{(j)}(u_i'') = \text{par}^{(j)}(v_i'')$ for each $j \in [2t]$. Thus it requires $O(t) = O(\log(\text{dep}(\text{par})))$ time. To conclude, $\text{LCA}(\text{par} : V \to V, Q)$ (Algorithm 14) can be implemented in the $(0, \delta)$-MPC model for any constant $\delta \in (0,1)$ in $O(\log(\text{dep}(\text{par})))$ parallel time. □

Lemma 5.3.4. *Let* $\text{par} : V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V. *Let* $Q = \{(u_1, v_1), (u_2, v_2), \cdots, (u_q, v_q)\} \subseteq V \times V$ *satisfy* $\forall j \in [q]$, v_j *is an ancestor (See Definition 5.2.8) of* u_j *in* par. *Let* $n = |V|, N = n + q$. MULTIPATHLARGE(par, Q) *(Algorithm 15) can be implemented in* $(\gamma, \delta)-$MPC *model for any* γ *with* $n \log \text{dep}(\text{par}) + \sum_{i=1}^{q}(\text{dep}_{\text{par}}(u_i) - \text{dep}_{\text{par}}(v_i) + 1) = O(N^{1+\gamma})$ *and any constant* $\delta \in (0,1)$ *in* $O(\text{dep}(\text{par}))$ *parallel running time.*

Proof. By Lemma 4.4.6, line 3 can be implemented in total space $O(n \log \text{dep}(\text{par}))$ and $O(\log(\text{dep}(\text{par})))$ parallel running time. It is easy to see that all the other steps in the procedure can be done by the operations shown in **Multiple queries**. Notice that after each round, we need to do load balancing (see **Load balance**) to make each machine have large enough available local memory. The

total space needed is to store all the pathes and the output of line 3. Notice that in round i, we do not need to keep $S_j^{(i')}$ for $i' < i-1$, thus, the space to keep $S_j^{(i)}$ for all $j \in [q]$ only needs $O(\sum_{j=1}^{q}(\mathrm{dep}_{\mathrm{par}}(u_j) - \mathrm{dep}_{\mathrm{par}}(v_j) + 1))$ space.

Thus, the total space needed is at most $O(n \log \mathrm{dep}(\mathrm{par}) + \sum_{i=1}^{q}(\mathrm{dep}_{\mathrm{par}}(u_i) - \mathrm{dep}_{\mathrm{par}}(v_i) + 1)) = O(N^{1+\gamma})$. The parallel running time is then $O(\mathrm{dep}(\mathrm{par}))$. □

Lemma 5.3.5. *Let* $\mathrm{par} : V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V. *Let* $Q = \{(u_1, v_1), (u_2, v_2), \cdots, (u_q, v_q)\} \subseteq V \times V$ *satisfy* $\forall j \in [q]$, v_j *is an ancestor (See Definition 5.2.8) of* u_j *in* par. *Let* $n = |V|, N = n + q$. MULTIPATH(par, Q) *(Algorithm 16) can be implemented in* $(0, \delta)$ − MPC *model for any constant* $\delta \in (0, 1)$ *in* $O(\mathrm{dep}(\mathrm{par}))$ *parallel running time.*

Proof. Consider the implementation of MULTIPATH$(\mathrm{par} : V \to V, Q)$ (Algorithm 16) in the MPC model. The steps before line 6 are the same as the steps of the LCA procedure described in Algorithm 14. According to Lemma 5.3.3, they can be implemented in the MPC model with $O(|V|) = O(n)$ total space and $\Theta(n^{\delta})$ local memory per machine for any constant $\delta \in (0, 1)$ in $O(\log(\mathrm{dep}(\mathrm{par})))$ time. In line 7, all queries $(u_i, v_i) \in Q$ can be handled simultaneously. In line 8, if $\mathrm{dep}_{\mathrm{par}}(u_i) \leq \mathrm{dep}_{\mathrm{par}}(v_i) + 2t$, the length of the path from u_i to v_i is at most $2t$, and thus $P(u_i, v_i)$ can be computed in $O(t) = O(\log(\mathrm{dep}(\mathrm{par})))$ time (see **Multiple queries**). In line 9, we can use $O(t) = O(\log(\mathrm{dep}(\mathrm{par})))$ time to find the minimum $j \in [2t]$ such that $\mathrm{par}^{(j)}(u_i) \in V'$ (see **Multiple queries**). Then we can apply exponential search to find v_i' by using $g_0, g_1, \cdots, g_t$ in $O(t) = O(\log(\mathrm{dep}(\mathrm{par})))$ time (see **Multiple queries**). According to Lemma 5.2.7, $|V'| \leq n/\log(\mathrm{dep}(p))$. In line 10, according to Algorithm 15, Lemma 5.2.14 and Lemma 5.3.4, each path $P'(u_i', v_i')$ in par' can be generated simultaneously in the MPC model with $O(|V'| \log |V'| + \sum_{i \in [q]} |P'(u_i', v_i')|) = O(n + \sum_{i \in [q]} |P(u_i, v_i)|)$ total space in $O(\log(\mathrm{dep}(\mathrm{par}'))) = O(\log(\mathrm{dep}(\mathrm{par})))$ time. Consider the initialization of $A = (a_1, a_2, \cdots, a_h)$ in line 11. Vertex a_1 should be u_i and a_h should be v_i. By Lemma 5.2.7, $\forall j \in [h-1]$, $\mathrm{dep}(a_j) - \mathrm{dep}(a_{j+1}) \leq 2t$. Thus, the number of repetitions in the final step is at most $O(t) = O(\log(\mathrm{dep}(\mathrm{par})))$. To conclude, MULTIPATH$(\mathrm{par} : V \to V, Q = \{(u_1, v_1), (u_2, v_2), \cdots, (u_q, v_q)\})$ can be implemented in the MPC model with total space linear in

$O(|V| + \sum_{i\in[q]} |P(u_i, v_i)|)$ and local memory size $\Theta(|V|^\delta)$ per machine for any constant $\delta \in (0,1)$ in $O(\log(\text{dep}(\text{par})))$ time. □

5.3.3 Leaf sampling

Lemma 5.3.6. *Let* par $: V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V*, and* par *has a unique root. Let* $n = |V|$*. Let* δ *be an arbitrary constant in* $(0,1)$*, and let* $m = \lceil n^\delta \rceil$*. Then* LEAFSAMPLING$(\text{par}, m, \delta)$ *(Algorithm 17) can be implemented in* $(0,\delta)-$MPC *model. Furthermore, with probability at least* $1 - 1/(100m^{5/\delta})$*, the parallel running time is at most* $O(\log \text{dep}(\text{par}))$.

Proof. To implement line 4, for each $v \in V$, we can add par(v) to a temporary set X. Then each v can check whether v is a leaf by checking whether v is in X, and this can be done by the operations shown in **Set membership** and **Multiple queries**.

To implement line 5, for each $v \in V$, we can add v to the set $\text{child}_{\text{par}}(\text{par}(v))$. Then rank can be computed by the operations shown in **Indexing elements in sets** and **Multiple queries**. For line 6, we can implement it on a single machine, since a single machine has local memory $\Theta(m)$. For line 7 to line 9, for each $x \in L$, we add x into S with probability p, where p can be computed by querying the size of L (see **Sizes of sets** and **Multiple queries**). Line 10 can be implemented by operation described in **Indexing elements in sets**, **Set membership**, and **Multiple queries**. By Lemma 4.4.2, line 11 can be implemented in total space $O(N)$ and $O(\log \text{dep}(\text{par}))$ parallel time. By Property 3 of Lemma 5.2.18, with probability at least $1 - 1/(100m^{5/\delta})$, $|S|^2 = O(m)$. Thus, Q can be stored on a single machine. By Lemma 5.3.3, line 15 can be implemented in total space $O(n + |Q|) = O(n)$ and in $O(\log \text{dep}(\text{par}))$ parallel time. Then line 17-21 can be implemented on a single machine.

Thus, the total space needed is at most $O(n)$. The parallel time is at most $O(\log \text{dep}(\text{par}))$ □

5.3.4 DFS sequence

Lemma 5.3.7. *Let* par : $V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V*, and* par *has a unique root. Let* $n = |V|$*. Let* δ *be an arbitrary constant in* $(0, 1)$*, and let* $m = \lceil n^{\delta} \rceil$*.* SUBDFS$(\text{par}, m, \delta)$ *(Algorithm 18) can be implemented in* $(0, \delta)$ − MPC *model. Furthermore, with probability at least* $1 - 1/(100m^{5/\delta})$*, the parallel running time is at most* $O(\log \text{dep}(\text{par}))$*.*

Proof. By Lemma 5.3.6, line 5 can be implemented in total space $O(n)$ and with probability at least $1 - 1/(100m^{5/\delta})$ has parallel running time $O(\log \text{dep}(\text{par}))$. By Lemma 5.3.3, line 7 can be implemented in total space $O(n)$ and in parallel running time $O(\log \text{dep}(\text{par}))$. Line 9 can be implemented by operation shown in **Multiple queries**. By Lemma 5.3.5, since all the paths are disjoint (except the first path and the last path intersecting on the root) and V has n vertices, line 10 can be implemented in $O(n)$ total space and in $O(\log \text{dep}(\text{par}))$ parallel running time. Loop in line 13 and Loop in line 16 can be implemented in parallel. Line 20 can be implemented by operations shown in **Indexing elements in sets** and **Multiple queries**. Now we describe the implementation of line 21. Firstly, we can standardize (see **Sequence standardizing**) the sequence A'. For each tuple ("A'", (j, u)), create a tuple ("$temp_u$", j). Thus, "$temp_u$" is a set which contains all the positions that u appeared. For each tuple ("$temp_u$", j), we query (see **Multiple queries**) the index i (see **Indexing elements in sets**) of j in set ("$temp_u$", j), and create a tuple ("pos", $((u, i), j)$). Thus, the desired mapping pos is stored in the system. The loop in line 24 is implemented in parallel. Line 25 can be implemented by the operations shown in **Set membership** and **Multiple queries**. Line 26 to line 28 can be implemented by the operation shown in **Multiple queries**. Finally, line 30 can be implemented by **Multiple queries** and **Sequence duplicating**.

The total space used in the procedure is at most $O(n)$. The parallel running time is $O(\log \text{dep}(\text{par}))$.

□

Theorem 5.3.8. *Let* par : $V \to V$ *be a set of parent pointers (See Definition 4.2.5) on a vertex set* V*, and* par *has a unique root. Let* $n = |V|, m = n^{\delta}$ *for some arbitrary constant* $\delta \in (0, 1)$*.* DFS(par, m) *(Algorithm 19) can be implemented in* $(0, \delta)$ − MPC *model. With probability at least*

0.99, *the parallel running time is* $O(\log(\mathrm{dep}(\mathrm{par})))$.

Proof. By Lemma 5.3.7, line 5 can be implemented in total space $O(n)$. With probability at least $1 - 1/(100n^5)$, the parallel running time is $O(\log(\mathrm{dep}(\mathrm{par})))$. Line 8-10 can be implemented by operations shown in **Set membership** and **Multiple queries**. By Lemma 4.4.2, line 11 can be implemented in $O(n)$ total space, and $O(\log \mathrm{dep}(\mathrm{par}))$ parallel running time. The loop in line 14 contains multiple tasks (see Section 2.3.6 **Multiple Tasks**), thus we can implement those tasks in parallel. By Lemma 5.3.7, line 17 can be implemented in total space $O(|V_i'(v)|)$. Furthermore, with probability at least $1 - 1/(100n^5)$, the parallel running time is $O(\log(\mathrm{dep}(\mathrm{par})))$. Thus, the total space needed for those tasks is at most $O(n)$. Line 19 can be implemented by operations shown in **Indexing elements in sets**, **Sequence insertion** and **Multiple queries**.

Thus, the total space needed is $O(n)$. By taking union bound over all the task SUBDFS, with probability at least 0.99, the parallel running time is $O(\log \mathrm{dep}(\mathrm{par}))$. □

Now we are able to conclude the following theorem.

Theorem 5.3.9. *For any* $\gamma \in [0,2]$ *and any constant* $\delta \in (0,1)$, *there is a randomized* (γ, δ) – MPC *algorithm (Algorithm 19) which can output a Depth-First-Search sequence for any tree graph* $G = (V, E)$ *in* $O(\min(\log D \cdot \log(1/\gamma'), \log n))$ *parallel time, where* $n = |V|$, D *is the diameter of* G, *and* $\gamma' = (1+\gamma)\log_n \frac{2N}{n^{1/(1+\gamma)}}$. *The success probability is at least* 0.98. *In addition, if the algorithm fails, then it will return* FAIL.

Proof. Firstly, by Theorem 4.4.12, we can find a rooted tree with depth at most $D^{O(\log(1/\gamma'))}$. Algorithm 19 can output the DFS sequence for a rooted tree.

The implementation and parallel time of Algorithm 19 is shown by Theorem 5.3.8. The correctness of Algorithm 19 is proved by Theorem 5.2.25. The success probability of Algorithm 19 is proved by Theorem 5.2.27. □

5.4 2-Edge connectivity and biconnectivity

Consider a connected undirected graph G with a vertex set V and an edge set E. In the 2-edge connectivity problem, the goal is to find all the bridges of G, where an edge $e \in E$ is called a bridge if its removal disconnects G. In the biconnectivity problem, the goal is to partition the edges into several groups $E_1, E_2, \cdots, E_k$, i.e., $E = \bigcup_{i=1}^{k} E_i, \forall i \neq j, E_i \cap E_j = \emptyset$, such that $\forall e \neq e' \in E$, e and e' are in the same group if and only if there is a simple cycle in G which contains both e and e'. A subgraph induced by an edge group E_i is called a biconnected component (block). In other words, the goal of the biconnectivity problem is to find all the blocks of G.

In this section, we describe the algorithms for both the 2-edge connectivity problem and the biconnectivity problem in the sequential setting. In later sections, we will discuss how to implement them in the MPC model.

5.4.1 2-Edge connectivity

The 2-edge connectivity problem is much simpler than the biconnectivity problem. We first compute a spanning tree of the graph. Only a tree edge can be a bridge. Then for any non-root vertex v, if there is no non-tree edge which crosses between the subtree of v and the outside of the subtree of v, then the tree edge which connects v to its parent is a bridge.

Algorithm 20 2-Edge Connectivity Algorithm

1: **procedure** BRIDGES($G = (V, E)$) ▷ $G = (V, E)$ is a connected undirected graph.
2: Compute a rooted spanning tree of G. The spanning tree is represented by a set of parent pointers $\text{par} : V \to V$. ▷ Algorithm 10,Algorithm 11.
3: Compute $\text{lev} : V \to \mathbb{Z}_{\geq 0}$: for each $v \in V$,

$$\text{lev}(v) \leftarrow \min\left(\text{dep}_{\text{par}}(v), \min_{w \in V \setminus \{\text{par}(v)\}:(v,w)\in E} \text{dep}_{\text{par}}(\text{the LCA of } (v, w))\right).$$

4: Compute the DFS sequence A of par. ▷ Algorithm 19.
5: Initialize $B \leftarrow \emptyset$. For each non-root vertex v, let a_i, a_j be the first and the last appearance of v in A respectively. If $\min_{k:i\leq k\leq j} \text{lev}(a_k) \geq \text{dep}_{\text{par}}(v)$, $B \leftarrow B \cup \{\{v, \text{par}(v)\}\}$. Output B.
6: **end procedure**

Lemma 5.4.1 (2-Edge connectivity)**.** *Consider an undirected graph $G = (V, E)$. Let B be the output of* BRIDGES(G) *(Algorithm 20). Then B is the set of all the bridges of G.*

Proof. Suppose $\{u, v\} \in E$ is not a bridge. If $\{u, v\}$ is a non-tree edge in par, then since B only contains tree edges, $\{u, v\} \notin B$. Otherwise, suppose $\text{par}(v) = u$. There must be a non-tree edge $(x, y) \in E$ such that x is in the subtree of v but y is not in the subtree of v. Thus, the LCA of (x, y) is not v, and it is an ancestor of v which means that the depth of the LCA of (x, y) is smaller than $\text{dep}_{\text{par}}(v)$. By line 3, we have $\text{lev}(x) < \text{dep}_{\text{par}}(v)$. Let a_i, a_j be the first and the last appearance of v in the DFS sequence of par. Since x is in the subtree of v, there exists $k \in \{i, i+1, \cdots, j\}$ such that $v = a_k$. By line 5, since $\text{lev}(a_k) < \text{dep}_{\text{par}}(v)$, $\{u, v\} \notin B$.

If $\{u, v\} \in E$ is a bridge, then (u, v) must be a tree edge in par, i.e., either $\text{par}(u) = v$ or $\text{par}(v) = u$. Suppose $\text{par}(v) = u$. Then for any non-tree edge $\{x, y\}$ with x in the subtree of v, y must also be in the subtree of v. Thus, the depth of the LCA of $\{x, y\}$ should be at least $\text{dep}_{\text{par}}(v)$. By line 3, for any x in the subtree of v, we have $\text{lev}(x) \geq \text{dep}_{\text{par}}(v)$. Let a_i, a_j be the first and the last appearance of v in the DFS sequence of par. Since all the vertices $a_i, a_{i+1}, \cdots, a_j$ are in the subtree of v, we have $\{u, v\} \in B$ by line 5. □

5.4.2 Biconnectivity

In this section, we will show a biconnectivity algorithm. It is a modification of the algorithm proposed by [50]. The high level idea is to construct a new graph G' based on the input graph G, and reduce the biconnectivity problem of G to the connectivity problem of G'. Since the running time of our connectivity algorithm depends on the diameter of the graph, we also give an analysis of the diameter of the graph G'.

Lemma 5.4.2 (Biconnectivity)**.** *Consider an undirected graph $G = (V, E)$. Let* $\text{col} : E \to V$ *be the output of* BICONN(G) *(Algorithm 21). Then $\forall e, e' \in E, e \neq e'$,* col *satisfies* $\text{col}(e) = \text{col}(e') \Leftrightarrow$ *there is a simple cycle in G which contains both e and e'. Furthermore, the diameter of the graph G' constructed by* BICONN(G) *is at most $O(\text{dep}(\text{par}) \cdot \text{bi-diam}(G))$, the number of vertices of G' is at most $|V|$, and the number of edges of G' is at most $|E|$.*

Algorithm 21 Biconnectivity Algorithm

1: **procedure** BICONN($G = (V, E)$) ▷ $G = (V, E)$ is a connected undirected graph.
2: Compute a rooted spanning tree of G. The spanning tree is represented by a set of parent pointers $\mathrm{par} : V \to V$. ▷ Algorithm 10,Algorithm 11.
3: Compute $\mathrm{lev} : V \to \mathbb{Z}_{\geq 0}$: for each $v \in V$,

$$\mathrm{lev}(v) \leftarrow \min\left(\mathrm{dep}_{\mathrm{par}}(v), \min_{w \in V \setminus \{\mathrm{par}(v)\}:(v,w)\in E} \mathrm{dep}_{\mathrm{par}}(\text{the LCA of } (v, w))\right).$$

4: Compute the DFS sequence A of par. ▷ Algorithm 19.
5: Let r be the root of par. Initialize $V' \leftarrow V \setminus \{r\}, E' \leftarrow \emptyset$.
6: For each $v \in V'$, let a_i, a_j be the first and the last appearance of v in A respectively. If $\min_{k\in\{i,i+1,\cdots,j\}} \mathrm{lev}(a_k) < \mathrm{dep}_{\mathrm{par}}(\mathrm{par}(v))$, $E' \leftarrow E' \cup \{\{v, \mathrm{par}(v)\}\}$.
7: For each $\{u, v\} \in E$, if neither u nor v is the LCA of (u, v) in par, $E' \leftarrow E' \cup \{\{u, v\}\}$.
8: Compute the connected components of $G' = (V', E')$. Let $\mathrm{col}' : V' \to V'$ be the coloring of the vertices in V' such that $\forall u', v' \in V'$, u', v' are in the same connected component in $G' \Leftrightarrow \mathrm{col}'(u') = \mathrm{col}'(v')$.
9: Initialize $\mathrm{col} : E \to V$. For each $e = \{u, v\} \in E$, if $\mathrm{dep}_{\mathrm{par}}(u) \geq \mathrm{dep}_{\mathrm{par}}(v)$, set $\mathrm{col}(e) \leftarrow \mathrm{col}'(u)$; otherwise, set $\mathrm{col}(e) \leftarrow \mathrm{col}'(v)$. Output $\mathrm{col} : E \to V$.
10: **end procedure**

Proof. Each $v \in V'$ corresponds to a tree edge $\{\mathrm{par}(v), v\} \in E$. Since $V' \subset V$, $|V'| \leq |V|$. By line 6 and line 7, each edge of G creates at most 1 edge of G'. Thus, $|E'| \leq |E|$.

Claim 5.4.3. *If* $\mathrm{dist}_{G'}(u, v) < \infty$*, i.e., vertices* $u, v \in V'$ *are in the same connected component of* G'*, then there is a simple cycle in* G *which contains both edges* $\{u, \mathrm{par}(u)\}$ *and* $\{v, \mathrm{par}(v)\}$.

Proof. Firstly, let us consider the case when $\{u, v\} \in E'$. If $\{u, v\}$ is added into E' by line 7, then there is a simple cycle in G:

$$(u, \mathrm{par}^{(1)}(u), \mathrm{par}^{(2)}(u), \cdots, \text{the LCA of } (u, v), \cdots, \mathrm{par}^{(2)}(v), \mathrm{par}^{(1)}(v), v, u).$$

Both edges $\{u, \mathrm{par}(u)\}$ and $\{v, \mathrm{par}(v)\}$ are in the such cycle. If $\{u, v\}$ is added into E' by line 6, then $u = \mathrm{par}(v)$. Let a_i, a_j be the first and the last appearance of v in A respectively. By line 6, there exists k with $i \leq k \leq j$ such that $\mathrm{lev}(a_k) < \mathrm{dep}_{\mathrm{par}}(v)$. Thus, there is a vertex x in the subtree of v such that $\mathrm{lev}(x) < \mathrm{dep}_{\mathrm{par}}(u)$. By line 3, there is an edge $\{x, y\} \in E$ such that the depth of the LCA of $\{x, y\}$ is smaller than $\mathrm{dep}_{\mathrm{par}}(u)$ which means that y is not in the subtree of u. In this case,

there is a simple cycle in G:

$$(x, \mathrm{par}^{(1)}(x), \mathrm{par}^{(2)}(x), \cdots, v, u, \mathrm{par}(u), \cdots, \text{the LCA of } (x, y), \cdots, \mathrm{par}^{(2)}(y), \mathrm{par}^{(1)}(y), y, x).$$

Since $u = \mathrm{par}(v)$, both edges $\{v, \mathrm{par}(v)\}$, $\{u, \mathrm{par}(u)\}$ are in such cycle.

Suppose $v, u \in V'$ are in the same connected component of G' and $\{v, \mathrm{par}(v)\}$, $\{u, \mathrm{par}(u)\}$ are in a simple cycle C_1 in G. Suppose $u, w \in V'$ are in the same connected component of G' and $\{u, \mathrm{par}(u)\}$, $\{w, \mathrm{par}(w)\}$ are in a simple cycle C_2 in G. Then, v and w are in the same connected component of G'. The symmetric difference of the edge set of C_1 and the edge set of C_2 should form another simple cycle C_3 in G which contains both edges $\{v, \mathrm{par}(v)\}$ and $\{w, \mathrm{par}(w)\}$. By induction on $\mathrm{dist}_{G'}(v, w)$, the claim holds. □

By Claim 5.4.3 and line 9, $\forall u, v \in V'$, if $\mathrm{col}(\{u, \mathrm{par}(u)\}) = \mathrm{col}(\{v, \mathrm{par}(v)\})$, then there should be a simple cycle in G which contains both edges $\{u, \mathrm{par}(u)\}$ and $\{v, \mathrm{par}(v)\}$. Consider an edge $\{u, v\} \in E$ such that neither u nor v is the LCA of (u, v), i.e., $\{u, v\}$ is a non-tree edge. Without loss of generality, suppose $\mathrm{dep}_{\mathrm{par}}(u) \geq \mathrm{dep}_{\mathrm{par}}(v)$. There is always a cycle in G:

$$(u, \mathrm{par}^{(1)}(u), \mathrm{par}^{(2)}(u), \cdots, \text{the LCA of } (u, v), \cdots, \mathrm{par}^{(2)}(v), \mathrm{par}^{(1)}(v), v, u),$$

which contains both edges $\{u, v\}, \{u, \mathrm{par}(u)\}$. By line 9, we have $\mathrm{col}(\{u, v\}) = \mathrm{col}(\{u, \mathrm{par}(u)\}) = \mathrm{col}'(u)$. Therefore, $\forall e_1, e_2 \in E$, there are always tree edges $e'_1, e'_2 \in E$ such that $\mathrm{col}(e'_1) = \mathrm{col}(e_1), \mathrm{col}(e'_2) = \mathrm{col}(e_2)$, e_1, e'_1 are either in a simple cycle in G or $e_1 = e'_1$, and e_2, e'_2 are either in a simple cycle in G or $e_2 = e'_2$. If $\mathrm{col}(e_1) = \mathrm{col}(e_2)$, then $\mathrm{col}(e'_1) = \mathrm{col}(e'_2)$ which implies that e'_1, e'_2 are either in a simple cycle in G or $e'_1 = e'_2$. Hence if $\mathrm{col}(e_1) = \mathrm{col}(e_2)$, then either there is a simple cycle in G which contains both e_1, e_2 or $e_1 = e_2$.

Next, let us show that if there is a simple cycle in G which contains both edges $e, e' \in E$, then $\mathrm{col}(e) = \mathrm{col}(e')$. An observation is that each non-tree edge $e = (u, v)$ (i.e., neither u nor v is the

LCA of (u, v) in par) defines a simple cycle C_e in G:

$$(u, \text{par}^{(1)}(u), \cdots, \text{the LCA of } (u, v), \cdots, \text{par}^{(1)}(v), v, u).$$

Claim 5.4.4. *For any simple cycle C_e defined by a non-tree edge $e = \{u, v\}$, there is a path P_e in G' such that P_e contains every vertex in C_e except the LCA of (u, v) in* par. *Furthermore, the length of P_e is at most* $2\,\text{dep}(\text{par})$.

Proof. Without loss of generality, we can assume $\text{dep}_{\text{par}}(u) \geq \text{dep}_{\text{par}}(v)$. If v is an ancestor of u, then the cycle C_e is

$$(u, \text{par}^{(1)}(u), \text{par}^{(2)}(u), \cdots, \text{par}^{(s)}(u), v, u)$$

for some $s \geq 1$. For each $j \in [s]$, u is in the subtree of $\text{par}^{(j-1)}(u)$. By line 6, since $\text{lev}(u) \leq \text{dep}_{\text{par}}(v) < \text{par}^{(j)}(u)$ for any $j \in [s]$, we have $\{\text{par}^{(j-1)}(u), \text{par}^{(j)}(u)\} \in E'$. Thus, there is a path P_e in G': $(u, \text{par}^{(1)}(u), \text{par}^{(2)}(u), \cdots, \text{par}^{(s)}(u))$. In this case, the length of P_e should be at most dep(par).

If v is not an ancestor of u, then the cycle C_e is

$$(u, \text{par}^{(1)}(u), \cdots, \text{par}^{(s_1)}(u), \text{the LCA of } (u, v), \text{par}^{(s_2)}(v), \cdots, \text{par}^{(1)}(v), v, u)$$

for some $s_1, s_2 \geq 1$. By the similar argument, $\forall j \in [s_1]$ the edge $\{\text{par}^{(j-1)}(u), \text{par}^{(j)}(u)\}$ ($\forall j' \in [s_2]$ the edge $\{\text{par}^{(j'-1)}(v), \text{par}^{(j')}(v)\}$) is added into E' by line 6. By line 7, $\{u, v\}$ is added into E'. Therefore, there is a path P_e in G':

$$(\text{par}^{(s_1)}(u), \text{par}^{(s_1-1)}(u), \cdots, \text{par}^{(1)}(u), u, v, \text{par}^{(1)}(v), \text{par}^{(2)}(v), \cdots, \text{par}^{(s_2)}(v)).$$

In this case, the length of P_e should be at most $2\,\text{dep}(\text{par}) - 1$. □

Notice that all the simple cycles defined by the non-tree edges formed a cycle basis of the cycle space of G, i.e., the edge set of any simple cycle in G can be represented by an xor sum of the edge sets of cycles $C_{e_1}, C_{e_2}, \cdots, C_{e_s}$ defined by some non-tree edges $e_1, e_2, \cdots, e_s \in E$ [23]. Consider

any two tree edges $\{u, \text{par}(u)\}, \{v, \text{par}(v)\} \in E$ contained by a simple cycle C. Let $e_1, e_2, \cdots, e_s \in E$ be all the non-tree edges in C. Then C can be represented by an xor sum of $C_{e_1}, C_{e_2}, \cdots, C_{e_s}$. Furthermore, $\forall i \in [s-1]$, C_{e_i} and $C_{e_{i+1}}$ should have a common tree edge. According to Claim 5.4.4, for each $i \in [s]$, we can find a path P_{e_i} in G' and $\forall j \in [s-1]$, P_{e_j} intersects $P_{e_{j+1}}$. Therefore, u and v are in the same connected component in G'. By line 9, $\text{col}((u, \text{par}(u))) = \text{col}'(u) = \text{col}'(v) = \text{col}((v, \text{par}(v)))$. Now consider a non-tree edge $e = (u, v) \in E$. Without loss of generality, we can assume $\text{dep}_{\text{par}}(u) \geq \text{dep}_{\text{par}}(v)$. A tree edge $\{u, \text{par}(u)\}$ is the simple cycle C_e defined by e. By line 9, we know that $\text{col}(e) = \text{col}'(u) = \text{col}(\{u, \text{par}(u)\})$. Therefore, we can conclude that $\forall e_1, e_2 \in E$, if there is a simple cycle in G which contains both e_1, e_2, then $\text{col}(e_1) = \text{col}(e_2)$.

The only thing remaining to prove is the diameter of G'. According to Claim 5.4.3, $\forall u, v \in V'$ with $\text{dist}_{G'}(u, v) < \infty$, there is a cycle C in G which contains both edges $\{u, \text{par}(u)\}$ and $\{v, \text{par}(v)\}$.

Claim 5.4.5. *$\forall u, v \in V'$, if there is a cycle in G which contains both edges $\{u, \text{par}(u)\}$, $(v, \text{par}(v))$, then there is a cycle C in G with length $O(\text{bi-diam}(G))$ which contains both edges $\{u, \text{par}(u)\}$, $\{v, \text{par}(v)\}$.*

Proof. By the definition of bi-diam(G), there is a cycle C_1 with length at most bi-diam(G) which contains both vertices u, v. If C_1 already contains both edges $\{u, \text{par}(u)\}$, $\{v, \text{par}(v)\}$, then we are done. Otherwise, suppose C_1 does not contain $\{u, p(u)\}$. There is an another cycle C_2 with length at most bi-diam(G) which contains both vertices $\text{par}(u), v$. We can regard C_2 as two disjoint paths from $\text{par}(u)$ to v. Thus at least one of the path does not contain the edge $\{u, \text{par}(u)\}$. Suppose this path is $(\text{par}(u), \cdots, x, \cdots, v)$ where x is the first vertex which appears in C_1, then we can combine the path $(u, \text{par}(u), \cdots, x)$ with the path obtained by removing the sub-path from u to x of C_1 to get a new cycle which contains both the edge $\{u, \text{par}(u)\}$ and v. The length of the new cycle is at most $2 \cdot$ bi-diam(G). We can do the similar operation to add edge $\{v, \text{par}(v)\}$ into the cycle. Thus, finally we will get a cycle which contains both $\{u, \text{par}(u)\}$, $\{v, \text{par}(v)\}$ with length at most $3 \cdot$ bi-diam(G). □

According to the above claim, we can find a cycle C in G which contains both edges $\{u, \text{par}(u)\}$,

$\{v, \text{par}(v)\}$ with length at most $O(\text{bi-diam}(G))$. It means that C can be represented by an xor sum of $s \leq O(\text{bi-diam}(G))$ basis cycles $C_{e_1}, C_{e_2}, \cdots, C_{e_s}$ defined by non-tree edges $e_1, e_2, \cdots, e_s$. Furthermore, $\forall i \in [s-1]$, C_i and C_{i+1} have at least one common tree edge. By Claim 5.4.4, we can find s paths $P_{e_1}, P_{e_2}, \cdots, P_{e_s}$ defined by $e_1, e_2, \cdots, e_s$ in G' such that $\forall i \in [s-1]$, P_{e_i} intersects $P_{e_{i+1}}$ at some vertex, and u, v are on some path P_{e_x}, P_{e_y} respectively. Thus, $\text{dist}_{G'}(u, v) \leq \sum_{i=1}^{s} |P_{e_i}| \leq s \cdot O(\text{dep(par)}) \leq O(\text{dep(par)} \cdot \text{bi-diam}(G))$, where the second inequality follows from Claim 5.4.4. To conclude, $\text{diam}(G') \leq O(\text{dep(par)} \cdot \text{bi-diam}(G))$. □

5.5 2-Edge connectivity and biconnectivity in MPC

In this section, we will discuss how to implement the 2-edge connectivity algorithm and the biconnectivity algorithm in the MPC model. Let us firstly introduce how to implement an subroutine called range minimum query (RMQ) in the MPC model.

5.5.1 Parallel range minimum query

Range Minimum Query (RMQ) problem is defined as the following: given a sequence of n numbers $a_1, a_2, \cdots, a_n$, the goal is to preprocess the sequence a to get a data structure such that for any query $(p, q), (p < q)$ we can efficiently find the element which is the minimum in $a_p, a_{p+1}, \cdots, a_q$. A classic method is to preprocess a sparse table f in $\log(n)$ number of iterations such that $\forall i \in [n], j \in [\lceil \log n \rceil] \cup \{0\}$, $f_{i,j} = \arg\min_{i \leq i' \leq \min(n, i+2^j-1)} a_i$. To answer query for (p, q), it just needs to return $\arg\min_{i \in \{f_{p,j^*}, f_{q-2^{j^*}+1, j^*}\}} a_i$ for $j^* = \lfloor \log(q - p + 1) \rfloor$. In this section, we firstly show a modified data structure. We will compute $\widehat{f}_{i,j} = \arg\min_{i \leq i' \leq \min(n, i+\lceil n^\delta \rceil^j - 1)} a_{i'}$ The Algorithm is shown in Algorithm 22. Then we show how to use $\widehat{f}$ to compute f in Algorithm 23. The total space needed to store f for Algorithm 23 is $O(n \log n)$. In Algorithm 24, we show how to improve the space to $O(n)$ and we can finally solve RMQ in linear total space.

Lemma 5.5.1. *Let $a_1, a_2, \cdots, a_n$ be a sequence of numbers, and $\delta \in (0, 1)$. Let $\{\widehat{f}_{p,q}\}$ be the output of* SPARSETABLE$^+(a_1, a_2, \cdots, a_n, \delta)$ *(Algorithm 22). Then $\forall p \in [n], q \in \{0\} \cup [\lceil 1/\delta \rceil]$, $\widehat{f}_{p,q} = \arg\min_{p \leq p' \leq \min(n, i+\lceil n^\delta \rceil^q - 1)} a_{p'}$.*

Algorithm 22 A Sparser Table for RMQ

1: **procedure** SPARSETABLE$^+(a_1, a_2, \cdots, a_n, \delta)$ ▷ Lemma 5.5.1.
2: ▷ Output: $\widehat{f}_{i,j}$ for $i \in [n], j \in \{0\} \cup [\lceil 1/\delta \rceil]$.
3: Initially, for all $i \in [n]$ let $\widehat{f}_{i,0} = i$. $\forall i > n, j \in \mathbb{Z}$, let $\widehat{f}_{i,j} = 0$, and let $a_0 = \infty$. Let $m = \lceil n^\delta \rceil$.
4: For $t \in [\lceil 1/\delta \rceil]$, let $S_t = \{x \mid \exists y \in [m-1], x = y \cdot m^t\}$.
5: **for** $l = 1 \to \lceil 1/\delta \rceil$ **do**
6: **for** $j = 0 \to \lceil n/m \rceil$ **do**
7: $i \leftarrow j \cdot m + 1$.
8: $z^*_{j,l} \leftarrow \arg\min_{z:t\in[l-1], x\in S_t, z=\widehat{f}_{j\cdot m+1+x,t}} a_z$.
9: **for** $i' = 0 \to \min(m-1, n-i)$ **do**
10: $T \leftarrow \{x \in \mathbb{Z} \mid i + i' \le x \le i + m - 1\} \cup \{x \in \mathbb{Z} \mid i + m^l \le x \le i + m^l + i' - 1\} \cup \{z^*_{j,l}\}$
11: $\widehat{f}_{i+i',l} = \arg\min_{z\in T} a_z$.
12: **end for**
13: **end for**
14: $l \leftarrow l + 1$.
15: **end for**
16: **return** $\widehat{f}_{i,j}$ for $i \in [n], j \in \{0\} \cup [\lceil 1/\delta \rceil]$.
17: **end procedure**

Proof. The proof is by induction on q. When $q = 0$, the statement obviously holds for all $\widehat{f}_{p,0}$. Suppose all $p \in [n], \widehat{f}_{p,0}, \widehat{f}_{p,1}, \cdots, \widehat{f}_{p,q-1}$ satisfy the property. The first observation is that the value of $\widehat{f}_{p,q}$ will be assigned in the procedure when $l = q, j = \lfloor (p-1)/m \rfloor, i' = (p-1) \mod m$. Then by line 8, $z^*_{j,l}$ will be the position of the minimum value in $a_{j\cdot m+m}, a_{j\cdot m+m+2}, \cdots, a_{j\cdot m+m^l-1}$ by our induction hypothesis. Then by line 11, $\widehat{f}_{i+i',l}$ will be the position of the minimum value in $a_{j\cdot m+i'+1}, a_{j\cdot m+i'+2}, \cdots, a_{j\cdot m+i'+m^l}$. Thus, Since $j \cdot m + i' + 1 = p$, $\widehat{f}_{p,q}$ satisfies the property. □

Lemma 5.5.2. *Let $A = (a_1, a_2, \cdots, a_n)$ be a sequence of numbers. Let δ be an arbitrary constant in $(0, 1)$.* SPARSETABLE$^+(a_1, a_2, \cdots, a_n, \delta)$ *(Algorithm 22) can be implemented in $(0, \delta)-$*MPC *model with $O(1)$ parallel running time.*

Proof. Let A be the sequence $(a_1, a_2, \cdots, a_n)$. The algorithm takes $O(1/\delta)$ rounds. m is the local space of a machine. There are $\Theta(n/m)$ machines each holds a consecutive $\Theta(m)$ elements of sequence A. Now consider the round l. Machine $j \in \{0\} \cup [\lceil n/m \rceil]$ needs to compute $\widehat{f}_{j\cdot m+1,l}, \widehat{f}_{j\cdot m+2,l}, \cdots, \widehat{f}_{j\cdot m+m-1,l}$. The number of queries machine j made in line 8 and line 11 is at most $\sum_{t=1}^{\lceil 1/\delta \rceil} |S_t| + 2m \le O(m/\delta) = O(m)$. Thus, there are total $O(n)$ queries. These queries can be answered simultaneously by operation shown in **Multiple queries** (see Section 2.3).

Thus, the total space needed is $O(n)$, and the parallel running time is $O(1)$. □

Algorithm 23 A Sparse Table for RMQ

procedure SPARSETABLE($a_1, a_2, \cdots, a_n, \delta$) ▷ Lemma 5.5.3.
  ▷ Output: $f_{i,j}$ for $i \in [n], j \in \{0\} \cup [\lceil \log n \rceil]$.
  Initially, for all $i \in [n]$ let $f_{i,0} = i$. $\forall i > n, j \in \mathbb{Z}$, let $f_{i,j} = 0$, and let $a_0 = \infty$. Let $m = \lceil n^\delta \rceil$.
  Let $\{\widehat{f}_{p,q} \mid p \in [n], q \in \{0\} \cup \lceil 1/\delta \rceil\} = \text{SPARSETABLE}^+(a_1, a_2, \cdots, a_n, \delta)$. ▷ Algorithm 22.
  Let all undefined $\widehat{f}_{p,q}$ be 0.
  for $t \in [\lceil \log n \rceil]$ **do**
    if $2^t \le m$ **then**
      $k_t \leftarrow -1$
      $S_t \leftarrow \emptyset$
    else
      $k_t \leftarrow \lfloor \log_m(2^t - m) \rfloor$
      $S_t \leftarrow \{x \mid x \in [2^t - m - m^{k_t} + 1]$ s.t. $x \equiv 1 \pmod{m^{k_t}}$ or $(2^t - m - x) \equiv -1 \pmod{m^{k_t}}\}$
    end if
  end for
  for $j = 0 \to \lceil n/m \rceil$ **do**
    for $t = 0 \to \lceil \log n \rceil$ **do**
      $i \leftarrow j \cdot m + 1$.
      $z^*_{j,t} \leftarrow \arg\min_{z: x \in S_t, z = \widehat{f}_{j \cdot m + m + x, k_t}} a_z$.
      for $i' = 0 \to \min(m - 1, n - i)$ **do**
        $T_1 \leftarrow \{x \in \mathbb{Z} \mid i + i' \le x \le \min(i + m - 1, i + i' + 2^t - 1)\}$
        $T_2 \leftarrow \{x \in \mathbb{Z} \mid \max(i + 2^t, i + i') \le x \le i + 2^t + i' - 1\}$
        $T \leftarrow T_1 \cup T_2 \cup \{z^*_{j,t}\}$
        $f_{i+i',t} = \arg\min_{z \in T} a_z$.
      end for
    end for
  end for
  return $f_{i,j}$ for $i \in [n], j \in \{0\} \cup [\lceil \log n \rceil]$.
end procedure

Lemma 5.5.3. *Let $a_1, a_2, \cdots, a_n$ be a sequence of numbers, and $\delta \in (0, 1)$. Let $\{f_{p,q}\}$ be the output of* SPARSETABLE$(a_1, a_2, \cdots, a_n, \delta)$ *(Algorithm 23). Then $\forall p \in [n], q \in \{0\} \cup [\lceil \log n \rceil]$, $f_{p,q} = \arg\min_{p \le p' \le \min(n, i + 2^q - 1)} a_{p'}$.*

Proof. Let $m = \lceil n^\delta \rceil$. By Lemma 5.5.1, $\forall x \in [n], y \in \{0\} \cup [\lceil 1/\delta \rceil]$, $\widehat{f}_{x,y} = \arg\min_{x \le x' \le \min(n, i + m^y - 1)} a_{x'}$. Thus, by the definition of S_t, we know $z^*_{j,t} = \arg\min_{j \cdot m + m + 1 \le z \le j \cdot m + 2^t} a_z$. An observation is that the value of $f_{p,q}$ will be assigned in the procedure when $t = q, j = \lfloor (p - 1)/m \rfloor, i' = (p - 1) \mod m$.

By line 23, we know

$$f_{p,q} = f_{i+i',t} = \arg \min_{z:i+i'+1\le z\le i+i'+2^t} a_z = \arg \min_{p\le p'\le \min(n,i+2^q-1)} a_{p'}.$$

□

Lemma 5.5.4. *Let $a_1, a_2, \cdots, a_n$ be a sequence of numbers. Let δ be an arbitrary constant in $(0,1)$.* SPARSETABLE$(a_1, a_2, \cdots, a_n, \delta)$ *(Algorithm 23) can be implemented in $(\gamma, \delta)-$*MPC *model for any $\gamma \ge \log\log n/\log n$ in $O(1)$ parallel time.*

Proof. By Lemma 5.5.2, line 4 can be implemented in $O(n)$ total space and $O(1)$ parallel time. The loop in line 15 is similar to Algorithm 22. Each machine j needs to compute $f_{j\cdot m+1,t}, \cdots, f_{j\cdot m+m-1,t}$ for all $t \in [\lceil \log n \rceil] \cup \{0\}$. The difference from Algorithm 22 is that, it can compute for all t at the same time since it only depends on the value of $\widehat{f}$. The number of queries made by each machine is $O(m \log n)$. Thus, the total number of queries is at most $O(n \log n)$. These queries can be answered simultaneously by operation shown in **Multiple queries** (see Section 2.3).

Thus, the total space needed is $O(n \log n)$, and the parallel running time is $O(1)$. □

Algorithm 24 Multiple RMQ Algorithm (space efficient)

1: **procedure** RMQ(A, Q) ▷ $A = (a_1, a_2, \cdots, a_n) \in \mathbb{Z}^n$ is a sequence and $Q = \{(l_1, r_1), (l_2, r_2), \cdots, (l_q, r_q)\}$ is a set, where $\forall i \in [q], l_i, r_i \in [n], l_i + \lceil \log n \rceil \le r_i$.
2: Set $t \leftarrow \lceil \log n \rceil$. Set $A' \leftarrow (a'_1, a'_2, \cdots, a'_{\lceil n/t \rceil})$, where

$$\forall i \in [\lceil n/t \rceil], a'_i \leftarrow \min_{j\in[n]:(i-1)\cdot t<j\le i\cdot t} a_j.$$

3: Initialize left : $[n] \to \mathbb{Z}$, right : $[n] \to \mathbb{Z}$. For each $i \in [n]$, find $j \in [\lceil n/t \rceil]$ such that $i \in ((j-1)t, jt]$. Set left$(i) \leftarrow \min_{k\in[n]\cap((j-1)t,i]} a_k$, right$(i) \leftarrow \min_{k\in[n]\cap[i,jt]} a_k$.
4: **for** $(l_i, r_i) \in Q$ **do**
5: Find the smallest $l'_i \ge l_i$ with $l'_i \mod t = 0$ and find the largest $r'_i \le r_i$ with $r'_i \mod t = 0$.
6: If $l'_i = r'_i$, set $m_i \leftarrow \infty$; otherwise $m_i \leftarrow \min_{l'_i/t+1\le j\le r'_i/t} a'_j$. .
7: Set rmq$((l_i, r_i)) \leftarrow \min(\text{right}(l_i), m_i, \text{left}(r_i))$.
8: **end for**
9: **end procedure**

Lemma 5.5.5 (Range minimum query)**.** *Let* $A = (a_1, a_2, \cdots, a_n) \in \mathbb{Z}^n$ *be a sequence of n numbers and* $Q = \{(l_1, r_1), (l_2, r_2), \cdots, (l_q, r_q)\}$ *where* $\forall i \in [q], l_i, r_i \in [n], l_i + \lceil \log n \rceil \leq r_i$*. Let* $\mathrm{rmq} : Q \to \mathbb{Z}$ *be the output of* $\mathrm{RMQ}(A, Q)$*. Then* $\forall (l_i, r_i) \in Q$*,* $\mathrm{rmq}((l_i, r_i)) = \min_{j \in [n] \cap [l_i, r_i]} a_j$*. In addition,* RMQ *can be implemented in the* $(0, \delta)$*-MPC model for any constant* $\delta \in (0, 1)$ *in* $O(1)$ *parallel time.*

Proof. Firstly, let us consider the correctness of $\mathrm{RMQ}(A, Q)$. Let $t = \lceil \log n \rceil$. For a query $(l_i, r_i) \in Q$, since $l_i + t \leq r_i$, the l_i', r_i' found by line 5 will satisfy $l_i' \leq r_i'$. If $l_i' = r_i'$, then $m_i = \infty$ and $\mathrm{rmq}((l_i, r_i)) = \min(\min_{l_i \leq j \leq l_i'} a_j, \min_{l_i' \leq j \leq r_i} a_j) = \min_{l_i \leq j \leq r_i} a_j$. Otherwise, by line 6, $m_i = \min_{l_i'+1 \leq j \leq r_i'} a_j$. By line 7, $\mathrm{rmq}((l_i, r_i)) = \min(\min_{l_i \leq j \leq l_i'} a_j, \min_{l_i'+1 \leq j \leq r_i'} a_j, \min_{r_i' \leq j \leq r_i} a_j) = \min_{l_i \leq j \leq r_i} a_j$.

Let us analyze the total space required and the parallel time for running $\mathrm{RMQ}(A, Q)$ in the MPC model. According to Theorem 2.3.1, the sorting takes $O(1)$ time and requires linear total space. Notice that $\delta \in (0, 1)$ is a constant and each machine has $\Theta(n^\delta)$ local memory. We can sort $a_1, a_2, \cdots, a_n$ by their indexes and $o(n)$ number of duplicates of some elements in A such that $a_{i \cdot n^\delta + 1}, \cdots, a_{(i+1) \cdot n^\delta}, a_{(i+1) \cdot n^\delta + 1}, \cdots, a_{(i+1) \cdot n^\delta + t}$ are on the i^{th} machine. Therefore, the first two steps of $\mathrm{RMQ}(A, Q)$ can be implemented in the MPC model with $O(n)$ total space and in time $O(1)$. For line 4, we can handle all the queries $(l_i, r_i) \in Q$ simultaneously. Line 5 only requires local computations. Line 6 needs to handle at most $|Q|$ RMQ on the sequence A'. Notice that $|A'| = O(|A|/t) = O(n/\log n)$. Due to Lemma 5.5.3, Lemma 5.5.4 and **Multiple queries** (see Section 2.3), this can be implemented in the MPC model with $O(|A'| \log |A'| + |Q|) = O(n + q)$ total space and $O(1)$ parallel time. Line 7 can be done in $O(1)$ time. To conclude, $\mathrm{RMQ}(A, Q)$ can be implemented in the $(0, \delta)$-MPC model for any constant $\delta \in (0, 1)$ and the parallel time is $O(1)$. □

5.5.2 MPC implementation of 2-edge connectivity and biconnectivity

The input is a connected undirected graph $G = (V, E)$. G has $|V| = n$ vertices and $|E| = m$ edges. Thus, the input size is $N = m + n$. Consider the (γ, δ)-MPC model for $\gamma \in [0, 2]$ and an arbitrary constant $\delta \in (0, 1)$. The total space in the system should be $\Theta(N^{1+\gamma})$ and the local memory

size of each machine is $\Theta(N^{\delta})$. Let D and D' be the diameter and bi-diameter of G respectively.

Theorem 5.5.6 (2-Edge connectivity in MPC)**.** *For any $\gamma \in [0,2]$ and any constant $\delta \in (0,1)$, there is a randomized (γ,δ)-MPC algorithm which outputs all the bridges of the graph G in $O\left(\log D \cdot \log \frac{\log n}{\log(N^{1+\gamma}/n)}\right)$ parallel time. The success probability is at least 0.97. If the algorithm fails, then it returns FAIL.*

Proof. In the first step of BRIDGES(G) (Algorithm 20), according to Theorem 4.4.12, with probability 0.98, the rooted spanning tree of G can be computed in the MPC model with total space $O(N^{1+\gamma})$ in $O(\log \mathrm{diam}(G) \cdot \log \log_{N^{1+\gamma}/n} n)$ time, and the depth of the spanning tree is at most $\mathrm{diam}(G)^{O(\log \log_{N^{1+\gamma}/n} n)}$. In line 3, to compute $\mathrm{lev}(v)$ for each $v \in V$, we can query the LCA of (v,w) in par for each edge $\{v,w\} \in E$. We can use our LCA algorithm (Algorithm 14) as the subroutine for this purpose. It takes the total space $O(m)$ and the running time $O(\log(\mathrm{dep}(\mathrm{par}))) = O(\log \mathrm{diam}(G) \cdot \log \log_{N^{1+\gamma}/n} n)$ (see Lemma 5.2.12, Lemma 5.3.3). In line 4, with probability at least 0.99, the DFS sequence can be computed using $O(n)$ total space in time $O(\log(\mathrm{dep}(\mathrm{par}))) = O(\log \mathrm{diam}(G) \cdot \log \log_{N^{1+\gamma}/n} n)$ (see Theorem 5.3.9). In line 5, we can use sorting (see Theorem 2.3.1) to find the first appearance a_i and the last appearance a_j in the DFS sequence of each vertex v, and $\min_{k \in \{i,i+1,\cdots,j\}} \mathrm{lev}(a_k)$ corresponds to a range minimum query. If the size of the subtree of v is at most $\log n$, the corresponding RMQ can be solved by local computation. Otherwise, we use our RMQ algorithm (see Algorithm 24) to handle the corresponding RMQ of v. By Lemma 5.5.5, this step only takes $O(1)$ time and requires $O(n)$ space. To conclude, BRIDGES(G) (Algorithm 20) only takes total space $O(N^{1+\gamma})$ and has parallel time $O(\log \mathrm{diam}(G) \cdot \log \log_{N^{1+\gamma}/n} n)$.

Since the correctness of BRIDGES(G) (Algorithm 20) is guaranteed by Lemma 5.4.1, we can conclude the proof. □

Theorem 5.5.7 (Biconnectivity in MPC)**.** *For any $\gamma \in [0,2]$ and any constant $\delta \in (0,1)$, there is a randomized (γ,δ)-MPC algorithm which outputs all the biconnected components of the graph G in $O\left(\log D \cdot \log^2 \frac{\log n}{\log(N^{1+\gamma}/n)} + \log D' \cdot \log \frac{\log n}{\log(N^{1+\gamma}/n)}\right)$ parallel time. The success probability is at least 0.95. If the algorithm fails, then it returns FAIL.*

Proof. The first several lines before line 6 of BICONN(G) (Algorithm 21) are the same as BRIDGES(G) (Algorithm 20). Thus, the success probability of these lines is at least 0.97. The total space used is at most $O(N^{1+\gamma})$ and the running time is at most $O(\log \mathrm{diam}(G) \cdot \log \log_{N^{1+\gamma}/n} n)$. Line 6 of BICONN($G$) (Algorithm 21) corresponds to the RMQ problem which is almost the same as line 5 of BRIDGES(G) (Algorithm 20). Thus, it takes $O(n)$ total space and $O(1)$ parallel time. Line 7 requires m LCA queries. We can run our LCA algorithm (Algorithm 14) for this step. It takes $O(m+n)$ space and $O(\log(\mathrm{dep}(\mathrm{par}))) = O(\log \mathrm{diam}(G) \cdot \log \log_{N^{1+\gamma}/n} n)$ time (see Lemma 5.2.12, Lemma 5.3.3). By Lemma 5.4.2, we have $\mathrm{diam}(G') \leq \mathrm{diam}(G)^{O(\log \log_{N^{1+\gamma}/n} n)} \cdot \text{bi-diam}(G)$. According to Theorem 4.4.12, with probability at least 0.98, the connected components of G' can be computed in line 8, the total space needed is $O(N^{1+\gamma})$, and the parallel running time is $O(\log \mathrm{diam}(G) \log^2 \log_{N^{1+\gamma}/n} n + \log \text{bi-diam}(G) \log \log_{N^{1+\gamma}/n} n)$. To conclude, the total space needed is at most $O(N^{1+\gamma})$, and the parallel running time is $O(\log \mathrm{diam}(G) \log^2 \log_{N^{1+\gamma}/n} n + \log \text{bi-diam}(G) \log \log_{N^{1+\gamma}/n} n)$.

Since the correctness of BICONN(G) (Algorithm 21) is guaranteed by Lemma 5.4.2, we can conclude the proof. □

5.6 Open ear decomposition

In this section, we show how to extend our biconnectivity algorithm to find an open ear decomposition of a biconnected graph.

Let $G = (V, E)$ be an undirected graph with $n = |V|$ vertices and $m = |E|$ edges. An ear decomposition of G is a partition of E into ordered disjoint subsets $E_1, \cdots, E_s$ such that the following holds:

1. E_1 forms a simple cycle in the graph. E_1 is called a root ear.
2. For $i \in [s] \setminus \{1\}$, E_i forms a simple path in the graph. $\forall i \in [s]$, E_i is called a ear.
3. For $i \in [s] \setminus \{1\}$, each end point of the path formed by E_i is also in the path (or cycle) formed by E_j for some $j \in [i-1]$.

4. For $i \in [s] \setminus \{1\}$, no internal vertex of the path formed by E_i is in any path (or cycle) formed by E_j for $j \in [i-1]$.

Furthermore, for $i \in [s] \setminus \{1\}$ if the end points of the path formed by E_i are different, the ear is *open*. Otherwise, it is *closed*. If every ear except the root ear is open, the such ear decomposition is an open ear decomposition.

In the open ear decomposition algorithm, the goal is to find a mapping $\text{col} : E \to \mathbb{Z}_{\geq 1}$ such that $\forall e \in E, \text{col}(e)$ denotes that e is in the ear $E_{\text{col}(e)}$, and $E_1, E_2, \cdots, E_s$ is an open ear decomposition of G.

According to [87], a graph G has an open ear decomposition if and only if G is biconnected. Thus, in this section, we only consider the graph which is biconnected. For graphs which are not biconnected, we can first run our biconnectivity algorithm to find each biconnected components and then find the open ear decomposition for each biconnected component.

The high level idea of our open ear decomposition algorithm is very similar to [88]. In particular, we first find a spanning tree of the input graph. Then we generate a proper ordering of non-tree edges, i.e., we give each non-tree edge a proper rank. For each non-tree edge, we create an ear. For each tree edge, find a non-tree edge with the smallest rank such that the tree edge is on the path between two end vertices of such non-tree edge, and put the tree edge into the ear containing such non-tree edge. Our process of generating the ordering of non-tree edges and the process of determining the ear of each tree edge are different from those in [88].

Open ear decomposition has many applications in other graph problems. We refer readers to [88] for more applications of open ear decomposition.

5.6.1 Open ear decomposition via a proper ordering of non-tree edges

Consider an n-vertex m-edge biconnected graph $G = (V, E)$ and let par be an arbitrary rooted spanning tree of G. Then there are $s = m - n + 1$ non-tree edges $e_1, e_2, \cdots, e_s$. For $i \in [s]$, let C_{e_i}

be the simple cycle defined by e_i, i.e., C_{e_i} denotes the cycle

$$(u, \text{par}^{(1)}(u), \text{par}^{(2)}(u), \cdots, \text{the LCA of } (u,v), \cdots, \text{par}^{(2)}(v), \text{par}^{(1)}(v), v, u),$$

where u, v are end points of e_i.

Lemma 5.6.1 (Open ear decomposition obtained by a good ordering of non-tree edges.)**.** *Suppose* $\forall i \in [s] \setminus \{1\}, \exists j \in [i-1]$ *there is at least one edge* $e \in E$ *which appears in both cycles* C_{e_i} *and* C_{e_j}*. Let* $E_1, E_1, \cdots, E_s \subseteq E$ *satisfy that* $\forall e \in E, e \in E_i \iff i = \min_{j \in [s]: e \text{ appears in } C_{e_j}} j$*. Then* $E_1, E_1, \cdots, E_s$ *is an open ear decomposition of* G*.*

Proof. Since G is biconnected, every edge $e \in E$ should be in some simple cycle. Since all the simple cycles defined by the non-tree edges are a cycle basis of G [23], each edge $e \in E$ must be appeared in C_{e_i} for some $i \in [s]$. Thus, $E_1, E_2, \cdots, E_s$ is a partition of E, i.e., $\forall e \in E$, there is a unique $i \in [s]$ such that $e \in E_i$.

It is easy to verify that $\forall i \in [s], E_1 \cup E_2 \cup \cdots \cup E_i$ is exactly the set of all edges that appeared in at least one cycle of $C_{e_1}, C_{e_2}, \cdots, C_{e_i}$. As a specific case, E_1 is exactly the set of all edges appeared in C_{e_1}.

$\forall i \in [s]$, define $V_i = \{u \in V \mid \exists v \in V, \{u, v\} \in E_1 \cup E_1 \cup \cdots \cup E_i\}$, i.e., V_i is the vertex set of all vertices that appeared as an end point of an edge of $E_1 \cup E_2 \cup \cdots \cup E_i$. Let $\forall i \in [s], P_i = E_1 \cup E_2 \cup \cdots \cup E_i \setminus \{e_1, e_2, \cdots, e_i\}$. Let graph $T_i = (V_i, P_i)$.

Claim 5.6.2. $\forall i \in [s], T_i$ *is a tree. Furthermore,* $\forall \{u, v\} \in P_i$*, either* $u = \text{par}(v)$ *or* $v = \text{par}(u)$*.*

Proof. By the construction of E_j for $j \in [s]$, an edge $\{u, v\}$ in $E_1 \cup E_2 \cup \cdots \cup E_i$ for $i \in [s]$ must appear in some cycle C_{e_k} for $k \leq i$. By the definition of $C_{e_1}, C_{e_2}, \cdots, C_{e_i}$, if $\{u, v\} \in P_i$, i.e., $\{u, v\} \notin \{e_1, e_2, \cdots, e_i\}$, we know that either $u = \text{par}(v)$ or $v = \text{par}(u)$. Since par is a rooted tree and each edge in T_i has a form $\{u, \text{par}(u)\}$, $\forall i \in [s], T_i$ does not contain any cycle.

It suffices to show that $\forall i \in [s], T_i$ is connected. The proof is by induction. Consider the case $i = 1$. We have $P_1 = E_1 \setminus \{e_1\}$ which contains all edges on the path obtained from deleting the

edge e_1 from the cycle C_{e_1}. Now suppose T_{i-1} is connected. By the construction of V_i, we know that $\forall u \in V_i \setminus V_{i-1}$, there is an edge $\{u, v\} \in E_i \setminus \{e_i\}$ which is on the path obtained from deleting e_i from C_{e_i}. By the construction of $P_i = E_1 \cup E_2 \cup \cdots \cup E_i \setminus \{e_1, e_2, \cdots, e_i\}$, we know that the path obtained from deleting the edge e_i from the cycle C_{e_i} is a subgraph of T_i. Thus, all endpoints of edges in $E_i \setminus \{e_i\}$ are connected in T_i. Since $\exists j \in [i-1]$, C_{e_i} and C_{e_j} share a common edge e', we know that e' is in the tree T_{i-1} and is also on the path obtained from deleting e_i from C_{e_i}. Since T_{i-1} is connected, we can conclude that T_i is also connected. □

As we mentioned previously, the edges of E_1 form a simple cycle C_{e_1}. Now we are going to show that $E_2, E_3, \cdots, E_s$ are open ears. The proof is by induction. We observe that $\forall i \in [s]$, E_i is the set of edges appeared on the cycle C_{e_i} but not on the cycles $C_{e_1}, C_{e_2}, \cdots, C_{e_{i-1}}$. Equivalently, it means that E_i is the set of edges appeared on the cycle C_{e_i} but not on the tree T_{i-1}. Since $\exists j \in [i-1]$, C_{e_i} and C_{e_j} share a common edge $e' = \{u', v'\}$, there is at least one pair of different vertices u', v' such that both of them appear in both C_{e_i} and T_{i-1}. Consider arbitrary two different vertices u and v which both appear on the cycle C_{e_i} and the tree T_{i-1}. The path between u and v in T_{i-1} is the same as the path between u and v in the tree represented by par because T_{i-1} is a induced subgraph of the tree represented by par. By the construction of C_{e_i}, this path is also a subgraph of the cycle C_{e_i}. Thus, all edges which are on both C_{e_i} and T_{i-1} are consecutive on the cycle C_{e_i}. Thus, E_i forms a path such that the end vertices of the path are different vertices in T_{i-1}, and none of the internal vertex of the path is in T_{i-1}. By the construction of T_{i-1}, we know that all vertices in T_{i-1} are on ears $E_1, E_2, \cdots, E_{i-1}$. Thus, by the definition of open ear, E_i must be an open ear. □

Next, in Algorithm 25, we show how to find a good ordering of non-tree edges.

Lemma 5.6.3 (Good ordering of non-tree edges)**.** *Consider an n-vertex m-edge undirected biconnected graph $G = (V, E)$ and a rooted spanning tree of G represented by a set of parent pointers* par $: V \to V$. *Let $e_1, e_2, \cdots, e_{m-n+1}$ be the output of* ORDERING$(G,$ par$)$ *(Algorithm 25). Then, $\forall i \in [m-n+1] \setminus \{1\}, \exists j \in [i-1]$, there exists an edge which is on both simple cycles defined by e_i and e_j in* par.

Algorithm 25 Good Ordering of Non-tree Edges

1: **procedure** ORDERING($G = (V, E)$, par : $V \to V$) ▷ $G = (V, E)$ is an n-vertex m-edge biconnected undirected graph, par is a set of parent pointers which represents an arbitrary rooted spanning tree of G.

2: Compute lev : $V \to \mathbb{Z}_{\geq 0}$, edg : $V \to V$: for each $v \in V$,

$$\mathrm{lev}(v) \leftarrow \min\left(\mathrm{dep}_{\mathrm{par}}(v), \min_{w \in V \setminus \{\mathrm{par}(v)\} : \{v,w\} \in E} \mathrm{dep}_{\mathrm{par}}(\text{the LCA of } (v,w))\right),$$

$$\mathrm{edg}(v) \leftarrow \begin{cases} v, & \mathrm{lev}(v) = \mathrm{dep}_{\mathrm{par}}(v); \\ w, & \text{otherwise, arbitrary } w \in V \setminus \{\mathrm{par}(v)\} : \{v,w\} \in E, \mathrm{lev}(v) = \mathrm{dep}_{\mathrm{par}}(\text{the LCA of } (v,w)). \end{cases}$$

3: Compute the DFS sequence A of par. ▷ Algorithm 19.

4: Let r be the root of par. Initialize $V' \leftarrow V \setminus \{r\}, E' \leftarrow \emptyset, \mathrm{map} : E' \to E$.

5: For each $v \in V'$, let a_i, a_j be the first and the last appearance of v in A respectively. Compute $k^* = \arg\min_{k \in \{i, i+1, \cdots, j\}} \mathrm{lev}(a_k)$. If $\mathrm{lev}(a_{k^*}) < \mathrm{dep}_{\mathrm{par}}(\mathrm{par}(v))$, $E' \leftarrow E' \cup \{\{v, \mathrm{par}(v)\}\}, \mathrm{map}(\{v, \mathrm{par}(v)\}) \leftarrow \{a_{k^*}, \mathrm{edg}(a_{k^*})\}$.

6: For each $\{u, v\} \in E$, if neither u nor v is the LCA of (u, v) in par, $E' \leftarrow E' \cup \{\{u, v\}\}, \mathrm{map}(\{u, v\}) \leftarrow \{u, v\}$.

7: Compute a rooted spanning tree of $G' = (V', E')$. The spanning tree is a set of parent pointers $\mathrm{par}' : V' \to V'$. ▷ Algorithm 10, Algorithm 11.

8: Compute the DFS sequence $A' = (a'_1, a'_2, \cdots, a'_{2n-3})$ of par′. ▷ Algorithm 19.

9: For each $e \in E$, initialize $\mathrm{ord}(e) \leftarrow \infty$.

10: For $i \in [2n-4], \mathrm{ord}(\mathrm{map}(\{a'_i, a'_{i+1}\})) \leftarrow i$. If there is multiple i that tries to update $\mathrm{ord}(\mathrm{map}(\{a'_i, a'_{i+1}\}))$, let $\mathrm{ord}(\mathrm{map}(\{a'_i, a'_{i+1}\}))$ take the minimum i.

11: Sort all non-tree edges $e = \{u, v\} \in E$, i.e., $u \neq \mathrm{par}(v)$ and $v \neq \mathrm{par}(u)$: $e_1, e_2, \cdots, e_{m-n+1}$ such that $\mathrm{ord}(e_i) \leq \mathrm{ord}(e_{i+1})$ for all $i \in [m-n]$.

12: **end procedure**

Proof. Observe that the graph G' constructed by Algorithm 25 is exactly the same as that constructed by Algorithm 21. Since G is a biconnected graph, G' is a connected graph according to Lemma 5.4.2. Thus, a rooted spanning tree par′ of G' can be found, i.e., par′ has a unique root. Since $V' = V \setminus \{r\}$ where r is the root of the rooted spanning tree par, G' is a graph on $|V'| = n - 1$ vertices. Thus, according to Fact 5.2.5, the length of the DFS sequence of par′ is $2n - 3$. Let $A' = (a'_1, a'_2, \cdots, a'_{2n-3})$ computed by line 8 of Algorithm 25 be the DFS sequence of par′. According to Definition 5.2.4, $\forall i \in [2n-4], \{a'_i, a'_{i+1}\} \in E'$.

Claim 5.6.4. *For each edge $e' = \{u', v'\} \in E'$, $\{u, v\} = \mathrm{map}(e')$ is a non-tree edge in* par *such that the cycle defined by $\{u, v\}$ contains both edges $\{u', \mathrm{par}(u')\}, \{v', \mathrm{par}(v')\} \in E$.*

Proof. Consider the case that $e' = \{u', v'\}$ is added into E' by line 5 of Algorithm 25. In this case,

either $u' = \text{par}(v')$ or $v' = \text{par}(u')$. Without loss of generality, let us assume $u' = \text{par}(v')$. According to Fact 5.2.5, a_{k^*} is in the subtree of v'. By the computation of $\text{lev}(a_{k^*})$ and $\text{edg}(a_{k^*})$ in line 2 of Algorithm 25, $\text{lev}(a_{k^*}) = \text{dep}_{\text{par}}(\text{the LCA of } (a_{k^*}, \text{edg}(a_{k^*}) \text{ in par}))$, and $\text{map}(e') = \{a_{k^*}, \text{edg}(a_{k^*})\} \in E$ is a non-tree edge in par. According to line 5, $\text{dep}_{\text{par}}(\text{the LCA of } (a_{k^*}, \text{edg}(a_{k^*})) \text{ in par}) < \text{dep}_{\text{par}}(u')$. Since a_{k^*} is in the subtree of v', both edges $\{v', u'\}, \{u', \text{par}(u')\}$ are on the path from a_{k^*} to the LCA of $(a_{k^*}, \text{edg}(a_{k^*}))$ in par. Thus, both edges $\{v', \text{par}(v')\}, \{u', \text{par}(u')\}$ are on the cycle defined by $\text{map}(e')$.

Consider the case $e' = \{u', v'\}$ is added into E' by line 6 of Algorithm 25. We have $\text{map}(e') = \{u', v'\}$. According to line 6 of Algorithm 25, neither u' nor v' is the LCA of (u', v') in par. Thus, both edges $\{v', \text{par}(v')\}, \{u', \text{par}(u')\}$ are on the cycle defined by $\text{map}(e')$. □

Let $e_1, e_2, \cdots, e_{m-n+1}$ be the output of $\text{ORDERING}(G, \text{par})$. We want to show that $\forall i \in [m - n + 1] \setminus \{1\}, \exists j \in [i - 1]$, there exists an edge which is on both simple cycles defined by e_i and e_j in par.

Consider the case that $\text{ord}(e_i) = \infty$. Suppose $e_i = \{u, v\}$. Since $\{u, v\}$ is a non-tree edge in par, either $\{u, \text{par}(u)\}$ or $(v, \text{par}(v))$ is in the cycle defined by $\{u, v\}$. Without loss of generality, we assume $\{u, \text{par}(u)\}$ is in the cycle defined by $\{u, v\}$ in par. Let a'_q be the first appearance of u in A'. Then $\{a'_q, a'_{q+1}\} \in E'$. Let $e = \text{map}(\{a'_q, a'_{q+1}\})$. We have $\text{ord}(e) \leq q < \text{ord}(e_i)$. According to Claim 5.6.4, the cycle defined by the non-tree edge e in par contains the edge $\{a'_q, \text{par}(a'_q)\} = \{u, \text{par}(u)\} \in E$. Thus, $\exists j < i$ such that $\text{ord}(e_j) = \text{ord}(e) \leq q < \text{ord}(e_i)$.

Consider the case that $\text{ord}(e_i) = i' < \infty$. Then $e_i = \text{map}(\{a'_{i'}, a'_{i'+1}\})$. If $i \neq 1$, then $i' > 1$. Let $e = \text{map}(\{a'_{i'-1}, a'_{i'}\})$. According to Claim 5.6.4, $\{a'_{i'}, \text{par}(a'_{i'})\} \in E$ is contained by both cycles defined by non-tree edges e and e_i separately. Furthermore, we have $\text{ord}(e) \leq i' - 1 < \text{ord}(e_i)$. Thus, $\exists j < i$ such that $\text{ord}(e_j) = \text{ord}(e) \leq q < \text{ord}(e_i)$. □

5.6.2 Segment coloring over trees

In the previous section, we show how to find a good ordering of non-tree edges. Let $s = m - n + 1$. Suppose the order of non-tree edges is $e_1, e_2, \cdots, e_s$. Then according to Lemma 5.6.1,

each non-tree edge e_i should be in the ear E_i. But since each tree edge can be in multiple cycles defined by non-tree edges, it is non-trivial to find the ear which should contain the tree edge.

We reduce the problem of determining the ear of each tree edge to the following segment coloring problem. Given a rooted tree represented by a set of parent pointers $\text{par} : V \to V$ and a set of tuples (segments) $Q \subseteq V \times \mathbb{Z}_{\geq 0} \times [s]$, we want to output $\text{col} : V \to [s] \cup \{\infty\}$ such that

$$\forall v \in V, \text{col}(v) = \min\left(\infty, \min_{(u,l,c)\in Q: u \text{ is in the subtree of } v, \text{dep}_{\text{par}}(u)-l\leq \text{dep}_{\text{par}}(v)} c\right).$$

In other word, for each tuple (segment) (u, l, c), we want to assign each ancestor of u within distance l a color c, and if a vertex v is assigned multiple colors, v only keeps the smallest color.

To determine the ear of each tree edge, we can create Q as the following. For each $e_i = \{u_i, v_i\}$, let z_i be the LCA of (u_i, v_i) in par. If $u_i \neq z_i$, we add a tuple $(u_i, \text{dep}_{\text{par}}(u_i) - \text{dep}_{\text{par}}(z_i) - 1, i)$ into Q. If $v_i \neq z_i$, we add a tuple $(v_i, \text{dep}_{\text{par}}(v_i) - \text{dep}_{\text{par}}(z_i) - 1, i)$ into Q. It is easy to verify that for each tree edge $\{v, \text{par}(v)\}$, $\text{col}(v) = \min_{j\in[s]:\{v,\text{par}(v)\} \text{ appears in } C_{e_j}} j$. Thus, the tree edge $\{v, \text{par}(v)\}$ should be in the ear $E_{\text{col}(v)}$.

Segment coloring for fixed length segments

First let us consider a special case: $\forall(u, l, c)$, l is the same and is the power of 2. In this case, we solve the segment coloring problem over par. The algorithm is shown in Algorithm 26.

Algorithm 26 Segment Coloring for Segments with Fixed Length

1: **procedure** ColorSameLen($\text{par} : V \to V, Q \subset V \times \mathbb{Z}_{\geq 0} \times [s], p \in \mathbb{Z}_{\geq 0}$) ▷ par represents an arbitrary rooted spanning tree of G, each segment has length 2^p.
2: For each $v \in V$, initialize $g^{(0)}(v) \leftarrow \text{par}(v), \text{col}^{(0)}(v) \leftarrow \min(\infty, \min_{(u,l,c)\in Q: u=v \text{ or } \text{par}(u)=v} c)$.
3: **for** $i := 1 \to p$ **do**
4: For each vertex $v \in V$, $\text{col}^{(i)}(v) \leftarrow \text{col}^{(i-1)}(v)$.
5: For each vertex $v \in V$, if $\text{col}^{(i-1)}(v) < \text{col}^{(i)}(g^{(i-1)}(v))$, update $\text{col}^{(i)}(g^{(i-1)}(v)) \leftarrow \text{col}^{(i-1)}(v)$. If $\text{col}^{(i)}(g^{(i-1)}(v))$ is updated by multiple v, take the minimum value $\text{col}^{(i-1)}(v)$.
6: For each vertex $v \in V$, $g^{(i)}(v) \leftarrow g^{(i-1)}(g^{(i-1)}(v))$.
7: **end for**
8: Return $\text{col} : V \to [s]$ where $\forall v \in V, \text{col}(v) \leftarrow \text{col}^{(p)}(v)$.
9: **end procedure**

Lemma 5.6.5 (Segment coloring for segments with fixed length)**.** *Consider an n-vertex rooted tree represented by a set of parent pointers* $\mathrm{par} : V \to V$*. Let* $p \in \mathbb{Z}_{\geq 0}$*. Let* $s \in \mathbb{Z}_{\geq 1}$*. Let* $Q \subseteq V \times \mathbb{Z}_{\geq 0} \times [s]$ *satisfy that* $\forall (u, l, c) \in Q, l = 2^p$*. Let* $\mathrm{col} : V \to [s]$ *be the output of* ColorSameLen(par, Q, p) *(Algorithm 26). Then,*

$$\forall v \in V, \mathrm{col}(v) = \min\left(\infty, \min_{(u,l,c)\in Q: u \text{ is in the subtree of } v, \mathrm{dep}_{\mathrm{par}}(u) - l \leq \mathrm{dep}_{\mathrm{par}}(v)} c\right).$$

Proof. Similar as the proof of Lemma 4.2.9, we can prove that $\forall i \in \{0, 1, \cdots, p\}, \forall v \in V, g^{(i)}(v) = \mathrm{par}^{(2^i)}(v)$ by induction.

Claim 5.6.6.

$$\forall v \in V, i \in \{0, 1, \cdots, p\}, \mathrm{col}^{(i)}(v) = \min\left(\infty, \min_{(u,l,c)\in Q: u \text{ is in the subtree of } v, \mathrm{dep}_{\mathrm{par}}(u) - 2^i \leq \mathrm{dep}_{\mathrm{par}}(v)} c\right).$$

Proof. The proof is by induction. Notice that

$$\forall v \in V, \mathrm{col}^{(0)}(v) = \min\left(\infty, \min_{(u,l,c)\in Q: \mathrm{par}(u) = v \text{ or } u = v} c\right).$$

Thus the claim holds for the base case $i = 0$.

Suppose the claim holds for $i - 1$ for every vertex $v \in V$. Then we have $\forall v \in V$,

$$\begin{aligned}
&\mathrm{col}^{(i)}(v) \\
&= \min\left(\mathrm{col}^{(i-1)}(v), \min_{u\in V: \mathrm{par}^{(2^{i-1})}(u) = v} \mathrm{col}^{(i-1)}(u)\right) \\
&= \min\left(\infty, \min_{\substack{(u,l,c)\in Q: \\ u \text{ is in the subtree of } v, \mathrm{dep}_{\mathrm{par}}(u) - 2^{i-1} \leq \mathrm{dep}_{\mathrm{par}}(v)}} c, \min_{\substack{(u,l,c)\in Q: \\ \exists u' \in V, u \text{ is in the subtree of } u', \mathrm{dep}_{\mathrm{par}}(u) - 2^{i-1} \leq \mathrm{dep}_{\mathrm{par}}(u'), \mathrm{par}^{(2^{i-1})}(u') = v}} c\right) \\
&= \min\left(\infty, \min_{(u,l,c)\in Q: u \text{ is in the subtree of } v, \mathrm{dep}_{\mathrm{par}}(u) - 2^i \leq \mathrm{dep}_{\mathrm{par}}(v)} c\right),
\end{aligned}$$

where the first step follows from the computation of $\mathrm{col}^{(i)}(v)$, the second step follows from the induction hypothesis, and the last step follows from that $\forall v \in V$,

$$\begin{aligned}
&\left\{u \in V \mid u \text{ is in the subtree of } v, \mathrm{dep}_{\mathrm{par}}(u) - 2^i \le \mathrm{dep}_{\mathrm{par}}(v)\right\} \\
=&\left\{u \in V \mid u \text{ is in the subtree of } v, \mathrm{dep}_{\mathrm{par}}(u) - 2^{i-1} \le \mathrm{dep}_{\mathrm{par}}(v)\right\} \\
&\cup \left\{u \in V \mid \exists u' \in V, u \text{ is in the subtree of } u', \mathrm{dep}_{\mathrm{par}}(u) - 2^{i-1} \le \mathrm{dep}_{\mathrm{par}}(u'), \mathrm{par}^{(2^{i-1})}(u') = u\right\}.
\end{aligned}$$

□

Since $\mathrm{col} \equiv \mathrm{col}^{(p)}$, the lemma is directly implied by the above claim since l is always 2^p. □

Segment coloring for segments with various lengths

. In this section, we show how to reduce the general segment coloring problem to the segment coloring problem for fixed length segments. The algorithm is shown in Algorithm 27.

Algorithm 27 Segment Coloring for Segments with Various Lengths

procedure ColorDifLen(par : $V \to V, Q \subset V \times \mathbb{Z}_{\ge 0} \times [s]$) ▷ par represents an arbitrary rooted spanning tree of G.
  Compute the depth of par, and set $d \leftarrow \mathrm{dep}(\mathrm{par}), t \leftarrow \lceil \log d \rceil$. Compute $\mathrm{par}^{(2^i)}(v)$ for each $v \in V, i \in \{0, 1, \cdots, t\}$ ▷ Run FindAncestors(par) (see Algorithm 6).
  Initialize $Q'_t \leftarrow \emptyset$. For each $(u, l, c) \in Q$, add $(u, \min(l, d), c)$ into Q'_t.
  For $u \in V, l \in \mathbb{Z}_{\ge 0}$ if there are multiple c s.t. $(u, l, c) \in Q'_t$, only keep $(u, l, c) \in Q'_t$ with the minimum c and delete others.
  for $i := t \to 0$ **do**
    Compute $Q_i = \{(u, l, c) \in Q'_i \mid l = 2^i\}$.
    Initialize $Q'_{i-1} \leftarrow \emptyset$. For each $(u, l, c) \in Q'_i$ with $l < 2^i$, if $l < 2^{i-1}$, add (u, l, c) into Q'_{i-1}, otherwise, add $(u, 2^{i-1}, c)$ and $(\mathrm{par}^{(2^{i-1})}(u), l - 2^{i-1}, c)$ into Q'_{i-1}.
    For $u \in V, l \in \mathbb{Z}_{\ge 0}$ if there are multiple c s.t. $(u, l, c) \in Q'_{i-1}$, only keep $(u, l, c) \in Q'_{i-1}$ with the minimum c and delete others.
  end for
  For each $v \in V$, initialize $\mathrm{col}_{-1}(v) \leftarrow \infty$. For each $(u, l, c) \in Q'_{-1}, \mathrm{col}_{-1}(u) \leftarrow c$.
  For each $i \in \{0, 1, \cdots, t\}$, find $\mathrm{col}_i \leftarrow$ColorSameLen(par, Q_i, i). ▷ See Algorithm 26.
  Initialize col : $V \to [s]$. For $v \in V$, $\mathrm{col}(v) \leftarrow \min_{i \in \{-1, 0, 1, \cdots, t\}} \mathrm{col}_i(v)$.
  Return col.
end procedure

Lemma 5.6.7 (Segment coloring for segments with various lengths)**.** *Consider an n-vertex rooted tree represented by a set of parent pointers par* $: V \to V$*. Let* $s \in \mathbb{Z}_{\geq 1}$*. Let* $Q \subseteq V \times \mathbb{Z}_{\geq 0} \times [s]$ *be a set of m tuples. Let* col *be the output of* ColorDifLen(par, Q) *(Algorithm 27). Then,*

$$\forall v \in V, \mathrm{col}(v) = \min\left(\infty, \min_{(u,l,c)\in Q: u \text{ is in the subtree of } v, \mathrm{dep}_{\mathrm{par}}(u)-l\leq \mathrm{dep}_{\mathrm{par}}(v)} c\right).$$

Furthermore, let $t, Q'_{-1}, Q'_0, Q'_1, \cdots, Q'_t$ *be the same as computed in Algorithm 27. Then* $\forall i \in \{-1, 0, 1, \cdots, t\}$, $|Q'_i| \leq m + n$*.*

Proof. Firstly, according to Lemma 4.3.6, line 2 can be implemented by calling Algorithm 6.

Let us first prove that $\forall i, |Q'_i| \leq m + n$.

Claim 5.6.8. $\forall i \in \{-1, 0, \cdots, t\}$*, we have* $\forall (u, l, c) \in Q'_i, l \leq 2^i$ *and* $|\{(u, l, c) \in Q'_i \mid l < 2^i\}| \leq m$*.*

Proof. The proof is by induction. For $i = t$, we have $|Q'_t| \leq |Q| = m$ and $\forall (u, l, c) \in Q'_t, l \leq \mathrm{dep(par)} \leq 2^t$. Thus the claim is true for $i = t$.

Suppose the claim is true for i. consider the construction of Q'_{i-1}. For each tuple $(u, l, c) \in Q'_i$, if $l = 2^i$, we will not create any tuple for Q'_{i-1}. For each tuple $(u, l, c) \in Q'_i$, if $l < 2^i$, we will create at most one tuple (u', l', c) for Q'_{i-1} such that $l' < 2^{i-1}$. Thus, by induction hypothesis $|\{(u, l, c) \in Q'_{i-1} \mid l < 2^{i-1}\}| \leq |\{(u, l, c) \in Q'_i \mid l < 2^i\}| \leq m$ Furthermore, for each tuple $(u, l, c) \in Q'_i$, if $l < 2^i$, any tuple (u', l', c) created satisfies that $l' \leq 2^{i-1}$. □

According to Claim 5.6.8, $\forall i \in \{-1, 0, \cdots, t\}, |Q'_i| \leq m + n$ since $\forall u \in V$, there is at most one tuple $(u, 2^i, c) \in Q'_i$ due to line 4 and line 8 of Algorithm 27. According to Claim 5.6.8, we know that $\forall (u, l, c) \in Q'_{-1}$, we have $l = 0$.

Next, let us consider col(v) for $v \in V$. According to line 4 and line 8, we know that $\forall v \in V$, $\mathrm{col}_{-1}(v) = c$ if there is a unique $(v, 0, c) \in Q'_{-1}$, otherwise $\mathrm{col}_{-1}(v) = \infty$. Thus, we have

$$\mathrm{col}_{-1}(v) = \min\left(\infty, \min_{(u,0,c)\in Q'_{-1}: u \text{ is in the subtree of } v, \mathrm{dep}_{\mathrm{par}}(u)-0\leq \mathrm{dep}_{\mathrm{par}}(v)} c\right). \tag{5.1}$$

Claim 5.6.9. *$\forall i \in \{-1, 0, \cdots, t\}$, we have $\forall v \in V$:*

$$\min_{(u,l,c)\in Q: u \text{ is in the subtree of } v, \mathrm{dep}_{\mathrm{par}}(u)-l\le \mathrm{dep}_{\mathrm{par}}(v)} c = \min_{(u,l,c)\in Q'_i \cup \bigcup_{j=i+1}^{t} Q_j : u \text{ is in the subtree of } v, \mathrm{dep}_{\mathrm{par}}(u)-l\le \mathrm{dep}_{\mathrm{par}}(v)} c.$$

Proof. The proof is by induction. Let us consider the case $i = t$. According to the construction of Q'_t, we can easily verify that $\forall v \in V, \widehat{c} \in [s]$:

$$\begin{aligned} &\exists (u, l, c) \in Q, u \text{ is in the subtree of } v, \mathrm{dep}_{\mathrm{par}}(u) - l \le \mathrm{dep}_{\mathrm{par}}(v), c \le \widehat{c} \\ \iff\ &\exists (u', l', c') \in Q'_t, u' \text{ is in the subtree of } v, \mathrm{dep}_{\mathrm{par}}(u') - l' \le \mathrm{dep}_{\mathrm{par}}(v), c' \le \widehat{c}. \end{aligned}$$

Thus, the claim holds for $i = t$. Now suppose the claim holds for i. To prove the claim for $i - 1$, according to the induction hypothesis, it suffices to prove $\forall v \in V, \widehat{c} \in [s]$,

$$\begin{aligned} &\exists (u, l, c) \in Q'_i, u \text{ is in the subtree of } v, \mathrm{dep}_{\mathrm{par}}(u) - l \le \mathrm{dep}_{\mathrm{par}}(v), c \le \widehat{c} \\ \iff\ &\exists (u', l', c') \in Q_i \cup Q'_{i-1}, u' \text{ is in the subtree of } v, \mathrm{dep}_{\mathrm{par}}(u') - l' \le \mathrm{dep}_{\mathrm{par}}(v), c' \le \widehat{c}. \end{aligned}$$

This can be easily verified by the construction of Q_i and Q'_{i-1}. Thus the claim holds for $i - 1$. □

Thus, we have $\forall v \in V$:

$$\begin{aligned} \mathrm{col}(v) &= \min(\mathrm{col}_{-1}(v), \mathrm{col}_0(v), \cdots, \mathrm{col}_t(v)) \\ &= \min\left(\infty, \min_{(u,l,c)\in Q'_{-1}\cup\bigcup_{j=0}^{t} Q_j : u \text{ is in the subtree of } v, \mathrm{dep}_{\mathrm{par}}(u)-l\le \mathrm{dep}_{\mathrm{par}}(v)} c\right) \\ &= \min\left(\infty, \min_{(u,l,c)\in Q: u \text{ is in the subtree of } v, \mathrm{dep}_{\mathrm{par}}(u)-l\le \mathrm{dep}_{\mathrm{par}}(v)} c\right), \end{aligned}$$

where the second step follows from Equation (5.1) and Lemma 5.6.5, and the last step follows from Claim 5.6.9. □

Next, we show how to improve the space usage of Algorithm 27, the improved space efficient algorithm is given in Algorithm 29. Before presenting Algorithm 29, we need to present a useful

subroutine which computes a target ancestor for each vertex (see Algorithm 28).

Algorithm 28 Find Target Ancestor for Each Vertex

procedure TARANCESTOR(par : $V \to V, Q \subseteq V \times \mathbb{Z}_{\geq 0}$) ▷ par represents an arbitrary rooted spanning tree of G. The goal is to compute $\text{par}^{(l)}(v)$ for each tuple $(v, l) \in Q$
    Compute the depth of each vertex in par. ▷ Run TREECONTRACTION($(V, \emptyset)$, par) (see Algorithm 3. Let the output $g^{(r)} : V \to \mathbb{Z}_{\geq 0}$ be $\text{dep}_{\text{par}} : V \to \mathbb{Z}_{\geq 0}$ (see Lemma 4.2.9).
    Set $d \leftarrow \text{dep}(\text{par}), t \leftarrow \lceil \log d \rceil$.
    Initialize $g^{(0)}(v) \leftarrow \text{par}(v)$ for each $v \in V$.
    Initilize $\text{map}^{(0)}(v, l) \leftarrow v$ for each tuple $(v, l) \in Q$.
    for $i = 0 \to t$ **do**
        For each tuple $(v, l) \in Q$, if $\min(\text{dep}_{\text{par}}(v), l) \bmod 2^{i+1} \geq 2^i$, $\text{map}^{(i+1)}(v, l) \leftarrow g^{(i)}(\text{map}^{(i)}(v, l))$, otherwise $\text{map}^{(i+1)}(v, l) \leftarrow \text{map}^{(i)}(v, l)$.
        For each vertex $v \in V$, $g^{(i+1)}(v) \leftarrow g^{(i)}(g^{(i)}(v))$.
    end for
    Return $\text{map}(v, l) \leftarrow \text{map}^{(t+1)}(v, l)$ for each $(v, l) \in Q$.
end procedure

Lemma 5.6.10. *Consider an n-vertex rooted spanning tree represented by a set of parent pointers* $\text{par} : V \to V$ *and a set of m tuples* $Q \subseteq V \times \mathbb{Z}_{\geq 0}$. *Let* map *be the output of* TARANCESTOR(par, Q) *(Algorithm 28). Then,* $\forall (v, l) \in Q, \text{map}(v, l) = \text{par}^{(l)}(v)$.

Proof. Similar as the proof of Lemma 4.2.9, we can prove that $\forall i \in \{0, 1, \cdots, t+1\}$, $\forall v \in V, g^{(i)}(v) = \text{par}^{(2^i)}(v)$ by induction. Notice that if $l \geq \text{dep}_{\text{par}}(v)$, then $\text{par}^{(l)}(v) = \text{par}^{(\text{dep}_{\text{par}}(v))}(v)$.

Now, consider an arbitrary tuple $(v, l) \in Q$. We want to show that $\forall i \in \{0, 1, \cdots, t+1\}$, $\text{map}^{(i)}(v, l) = \text{par}^{(\min(l, \text{dep}_{\text{par}}(v)) \bmod 2^i)}(v)$. The proof is by induction. If $i = 0$, we have $\min(l, \text{dep}_{\text{par}}(v)) \bmod 1 = 0$ and $\text{map}^{(0)}(v, l) = v = \text{par}^{(0)}(v, l)$. Now suppose the induction claim is true for i. If $\min(l, \text{dep}_{\text{par}}(v)) \bmod 2^{i+1} < 2^i$, then we have $\min(l, \text{dep}_{\text{par}}(v)) \bmod 2^i = \min(l, \text{dep}_{\text{par}}(v)) \bmod 2^{i+1}$ and $\text{map}^{(i+1)}(v, l) = \text{map}^{(i)}(v, l) = \text{par}^{((\min(l, \text{dep}_{\text{par}}(v)) \bmod 2^{i+1})}(v, l)$. If $\min(l, \text{dep}_{\text{par}}(v)) \bmod 2^{i+1} \geq 2^i$, then we have $\min(l, \text{dep}_{\text{par}}(v)) \bmod 2^{i+1} = (\min(l, \text{dep}_{\text{par}}(v)) \bmod 2^i) + 2^i$ and $\text{map}^{(i+1)}(v, l) = g^{(i)}(\text{map}^{(i)}(v, l)) = \text{par}^{((\min(l, \text{dep}_{\text{par}}(v)) \bmod 2^{i+1})}(v, l)$.

Notice that $\forall v \in V, \min(l, \text{dep}_{\text{par}}(v)) \leq 2^t$. Thus $\min(l, \text{dep}_{\text{par}}(v)) \bmod 2^{t+1} = \min(l, \text{dep}_{\text{par}}(v))$ which implies that $\text{map}(v, l) = \text{map}^{(t+1)}(v, l) = \text{par}^{(l)}(v)$. □

Algorithm 29 Segment Coloring for Segments with Various Lengths (space efficient)

procedure COLORING(par : $V \to V, Q \subset V \times \mathbb{Z}_{\geq 0} \times [s]$) ▷ par represents an arbitrary rooted spanning tree of G.
  $(V', \mathrm{par}') \leftarrow$ COMPRESS(par). ▷ Algorithm 12.
  Compute the depth of each vertex in par. ▷ Run TREECONTRACTION($(V, \emptyset)$, par) (see Algorithm 3). Let the output $g^{(r)} : V \to \mathbb{Z}_{\geq 0}$ be $\mathrm{dep}_{\mathrm{par}} : V \to \mathbb{Z}_{\geq 0}$ (see Lemma 4.2.9).
  Compute the depth of each vertex in par′. ▷ Run TREECONTRACTION($(V', \emptyset)$, par′) (see Algorithm 3). Let the output $g^{(r)} : V' \to \mathbb{Z}_{\geq 0}$ be $\mathrm{dep}_{\mathrm{par}'} : V \to \mathbb{Z}_{\geq 0}$ (see Lemma 4.2.9).
  Set $d \leftarrow \mathrm{dep}(\mathrm{par}), t \leftarrow \lceil \log d \rceil$.
  Compute mappings $g_0, g_1, \cdots g_t : V' \to V'$ such that $\forall v \in V', j \in \{0, 1, \cdots, t\}$, $g_j(v) = \mathrm{par}'^{(2^j)}(v)$. ▷ Run FINDANCESTORS(par) (see Algorithm 6).
  For each $(u, l, c) \in Q$, if $l > \mathrm{dep}_{\mathrm{par}}(u)$, replace (u, l, c) with $(u, \mathrm{dep}_{\mathrm{par}}(u), c)$ in Q.
  Initialize $Q' \leftarrow \emptyset$.
  Initialize $\mathrm{col}(v) \leftarrow \infty$ for $v \in V$.
  for $(u, l, c) \in Q$ **do**
    if $l \leq 10 \cdot t$ **then**
      Update $\mathrm{col}(v) \leftarrow c$ for every $v = \mathrm{par}^{(i)}(u), i \in \{0, 1, \cdots, l\}$ if $c < \mathrm{col}(v)$ ($\mathrm{col}(v)$ is updated by the minimum c).
    else
      Find the minimum $j \in \{0, 1 \cdots, 2t\}$ such that $x = \mathrm{par}^{(j)}(u) \in V'$.
      Let h be the maximum value such that $(\mathrm{dep}_{\mathrm{par}}(u) - h) \bmod t = 0$ and $h \leq l$.
      Find $y = \mathrm{par}^{(h)}(u)$. ▷ Run Algorithm 28 for par and all such tuples (u, h).
      $Q' \leftarrow Q' \cup \{(x, \mathrm{dep}_{\mathrm{par}'}(x) - \mathrm{dep}_{\mathrm{par}'}(y) - 1, c)\}$.
      Update $\mathrm{col}(v) \leftarrow c$ for every $v = \mathrm{par}^{(k)}(u), k \in \{0, 1, \cdots, j\}$ if $c < \mathrm{col}(v)$ ($\mathrm{col}(v)$ is updated by the minimum c).
      Update $\mathrm{col}(v)$ for every $v = \mathrm{par}^{(k)}(u), k \in \{h, h+1, \cdots, l\}$ if $c < \mathrm{col}(v)$ ($\mathrm{col}(v) \leftarrow c$ is updated by the minimum c).
    end if
  end for
  $(\mathrm{col}' : V' \to [s]) \leftarrow$ COLORDIFLEN(par′, Q'). ▷ Algorithm 27.
  For each $u \in V'$, update $\mathrm{col}(v) \leftarrow \mathrm{col}'(u)$ for every $v = \mathrm{par}^{(i)}(u), i \in \{0, 1, \cdots, t\}$ if $\mathrm{col}'(u) < \mathrm{col}(v)$ ($\mathrm{col}(v)$ is updated by the minimum $\mathrm{col}'(u)$).
  Return col.
end procedure

Lemma 5.6.11 (Segment coloring)**.** *Consider an n-vertex rooted tree represented by a set of parent pointers* par : $V \to V$. *Let $s \in \mathbb{Z}_{\geq 1}$. Let $Q \subseteq V \times \mathbb{Z}_{\geq 0} \times [s]$ be a set of m tuples. Let* col *be the output of* COLORING(par, Q) *(Algorithm 29). Then,*

$$\forall v \in V, \mathrm{col}(v) = \min\left(\infty, \min_{(u,l,c) \in Q : u \text{ is in the subtree of } v, \mathrm{dep}_{\mathrm{par}}(u) - l \leq \mathrm{dep}_{\mathrm{par}}(v)} c\right).$$

Proof. Consider $v \in V$. We first prove that $\mathrm{col}(v)$ computed by Algorithm 29 is at most c for every

$(u, l, c) \in Q$ satisfying that u is in the subtree of v and $\text{dep}_{\text{par}}(u) - l \leq \text{dep}_{\text{par}}(v)$. Let us focus on a tuple $(u, l, c) \in Q$ satisfying that u is in the subtree of v and $\text{dep}_{\text{par}}(u) - l \leq \text{dep}_{\text{par}}(v)$. In the first case $l \leq 10t$. In this case, $\text{col}(v) \leq c$ according to line 12. In the second case $l > 10t$. According to Lemma 5.2.7, both x and y are in V'. Furthermore y is an ancestor of x, x is an ancestor of u and $\text{dep}_{\text{par}}(u) - \text{dep}_{\text{par}}(x) \leq 2t, l - (\text{dep}_{\text{par}}(u) - \text{dep}_{\text{par}}(y)) \leq t$. If v is an ancestor of u but not an ancestor of x, we have $\text{col}(v) \leq c$ according to line 18. If v is an ancestor of y, we have $\text{col}(v) \leq c$ according to line 19. If v is an ancestor of x but not an ancestor of y, we can find a vertex $w \in V'$ such that $v = \text{par}^{(k)}(w)$ for some $k \in \{0, 1, \cdots, t\}$. According to line 17, line 22 and Lemma 5.6.7, we have $\text{col}'(w) \leq c$. According to line 23, we have $\text{col}(v) \leq \text{col}'(w) \leq c$.

Next, we prove that if $\text{col}(v) < \infty$, there exists $(u, l, c) \in Q$ satisfying that u is in the subtree of v and $\text{dep}_{\text{par}}(u) - l \leq \text{dep}_{\text{par}}(v)$ such that $c = \text{col}(v)$. Suppose $\text{col}(v)$ is finally updated by line 23, then there is a vertex $w \in V'$ such that $\exists k \in \{0, 1, \cdots, t\}, \text{par}^{(k)}(w) = v$ and $\text{col}(v) = \text{col}'(w)$. According to line 17, line 22 and Lemma 5.6.7, there is a tuple $(u, l, c) \in Q$ such that w is an ancestor of u, $\text{dep}_{\text{par}}(u) - l \leq \text{dep}_{\text{par}}(w) - t$ such that $\text{col}'(w) = c$. It implies that (u, l, c) satisfies that u is in the subtree of v and $\text{dep}_{\text{par}}(u) - l \leq \text{dep}_{\text{par}}(v)$.

Suppose $\text{col}(v)$ is finally updated by line 12, line 18 or line 19. Then, it is easy to verify that $\exists (u, l, c) \in Q$ such that u is in the subtree of v, $\text{dep}_{\text{par}}(u) - l \leq \text{dep}_{\text{par}}(v)$, and $c = \text{col}(v)$. □

5.6.3 Open ear decomposition

In this section, we present our final open ear decomposition algorithm (see Algorithm 30).

Lemma 5.6.12 (Open ear decomposition)**.** *Consider an n-vertex m-edge undirected biconnected graph $G = (V, E)$. Let $E_1, E_2, \cdots, E_s$ be the output of* OpenEarDecomp(G) *(Algorithm 30). Then $E_1, E_2, \cdots, E_s$ is an open ear decomposition of G.*

Proof. Notice that the cycle C_{e_i} defined by $e_i = \{u_i, v_i\}$ is:

$$(u_i, \text{par}(u_i), \text{par}^{(2)}(u_i), \cdots, z_i, \cdots, \text{par}^{(2)}(v_i), \text{par}(v_i), v_i, u_i),$$

Algorithm 30 Open Ear Decomposition

1: **procedure** OPENEARDECOMP($G = (V, E)$) ▷ G is an n-vertex m-edge biconnected graph.
2: Compute a rooted spanning tree of G. The spanning tree is represented by a set of parent pointers par : $V \to V$. ▷ Algorithm 10,Algorithm 11.
3: Let $e_1, e_2, \cdots, e_s$ be the output of ORDERING(G, par), where $s = m - n + 1$. ▷ Algorithm 25.
4: Let $E_i \leftarrow \{e_i\}$ for $i \in [s]$.
5: Initialize $Q \leftarrow \emptyset$. For each $e_i = \{u_i, v_i\}, i \in [s]$, find the LCA z_i of (u_i, v_i).
6: Compute $\mathrm{dep}_{\mathrm{par}}(v)$ for each $v \in V$. ▷ Run TREECONTRACTION($(V, \emptyset)$, par). Let the output $g^{(r)} : V \to \mathbb{Z}_{\geq 0}$ be $\mathrm{dep}_{\mathrm{par}} : V \to \mathbb{Z}_{\geq 0}$ (see Lemma 4.2.9).
7: Consider each $i \in [s]$. If $u_i \neq z_i$, we add a tuple $(u_i, \mathrm{dep}_{\mathrm{par}}(u_i) - \mathrm{dep}_{\mathrm{par}}(z_i) - 1, i)$ into Q. If $v_i \neq z_i$, we add a tuple $(v_i, \mathrm{dep}_{\mathrm{par}}(v_i) - \mathrm{dep}_{\mathrm{par}}(z_i) - 1, i)$ into Q. .
8: col $\leftarrow$COLORING(par, Q). ▷ Algorithm 29.
9: $\forall v \in V$ if $\mathrm{par}(v) \neq v$, add $\{v, \mathrm{par}(v)\}$ into $E_{\mathrm{col}(v)}$.
10: Output $E_1, E_2, \cdots, E_s$.
11: **end procedure**

where z_i is the LCA of (u_i, v_i).

According to Lemma 5.6.3, $\forall i \in [s] \setminus \{1\}, \exists j \in [i-1]$, there exists an edge which is on both simple cycles defined by e_i and e_j in par. According to line 4, line 9, line 7, and Lemma 5.6.11, we have

$$\forall i \in [s], E_i = \left\{ e \in E \mid i = \min_{j \in [s]: e \text{ appears in } C_{e_j}} j \right\}.$$

Thus, according to Lemma 5.6.1, $E_1, E_2, \cdots, E_s$ is an open ear decomposition of G. □

5.7 Open ear decomposition in MPC

In this section, we will discuss how to implement the open ear decomposition algorithm in the MPC model. For basic MPC operations, we refer readers to Section 2.3.

5.7.1 Find a proper ordering of non-tree edges in MPC

In this section, we show how to implement Algorithm 25 in the MPC model.

Lemma 5.7.1. *Let $G = (V, E)$ be an n-vertex m-edge undirected graph. Let* par : $V \to V$ *be a set of parent pointers representing a rooted spanning tree of G.* ORDERING(G, par) *(Algorithm 25)*

can be implemented in the (γ, δ)*-MPC model for any* $\gamma \in [0, 2]$ *and any constant* $\delta \in (0, 1)$ *in* $O\left((\log D' + \log \mathrm{dep}(\mathrm{par})) \cdot \log \frac{\log n}{\log(N^{1+\gamma}/n)}\right)$ *parallel time with probability at least* 0.96*, where* $D' =$ bi-diam(G').

Proof. Let $N = n + m$ be the input size. In line 2, to compute $\mathrm{lev}(v)$ and $\mathrm{edg}(v)$ for each $v \in V$, we can query the LCA of (v, w) in par for each edge $\{v, w\} \in E$. We can use our LCA algorithm (Algorithm 14) as the subroutine for this purpose. It takes the total space $O(N)$ and the running time $O(\log(\mathrm{dep}(\mathrm{par})))$ (see Lemma 5.2.12 and Lemma 5.3.3). In line 3, with probability at least 0.99, the DFS sequence can be computed using $O(n)$ total space in time $O(\log(\mathrm{dep}(\mathrm{par})))$ (see Theorem 5.3.9). In line 5, we can use sorting (see Theorem 2.3.1) to find the first appearance a_i and the last appearance a_j in the DFS sequence of each vertex v, and k^* corresponds to a range minimum query[1]. If the size of the subtree of v is at most $\log n$, the corresponding RMQ can be solved by local computation. Otherwise, we use our RMQ algorithm (see Algorithm 24) to handle the corresponding RMQ of v. By Lemma 5.5.5, this step only takes $O(1)$ time and requires $O(n)$ space. Line 6 can be implemented using $O(N)$ space and $O(1)$ parallel time. Since the graph G' computed by Algorithm 25 is the same as that computed by Algorithm 21. According to Lemma 5.4.2, we have $\mathrm{diam}(G') = O(\mathrm{dep}(\mathrm{par}) \cdot \text{bi-diam}(G))$, $|V'| = n - 1, |E'| \leq m$. According to Theorem 4.4.12, with probability at least 0.98, the rooted spanning tree par′ of G' in line 7 can be computed in the MPC model with total space $O(N^{1+\gamma})$ in $O(\log \mathrm{diam}(G') \cdot \log \log_{N^{1+\gamma}/n} n)$ parallel time, and the depth of the spanning tree par′ is at most $\mathrm{diam}(G')^{O(\log \log_{N^{1+\gamma}/n} n)}$. The DFS sequence of par′ can be computed in the MPC model with total space $O(n)$ in time $O(\log(\mathrm{dep}(\mathrm{par}')))$ parallel time with probability at least 0.99 (see Theorem 5.3.9). The remaining steps of the algorithm can be implemented by **Multiple queries** (see Section 2.3) and sorting (see Theorem 2.3.1). Thus, the remaining steps only need $O(N)$ total space and $O(1)$ parallel time. Thus, the overall total space needed is $O(N^{1+\gamma})$. The success probability is at least 0.96. The overall parallel time needed is $O\left((\log D' + \log \mathrm{dep}(\mathrm{par})) \cdot \log \frac{\log n}{\log(N^{1+\gamma}/n)}\right)$ where $D' = \text{bi-diam}(G')$. □

[1]Notice that this is doable since we can embed the index k into a_k, i.e., replace each a_k with $a_k \cdot 100N + k$. Thus $\lfloor a_k / 100N \rfloor = k$.

5.7.2 Segment coloring in MPC

In this section, we show how to implement Algorithm 26, Algorithm 27, Algorithm 28 and Algorithm 29.

Lemma 5.7.2. *Consider an n-vertex rooted tree represented by a set of parent pointers* $\mathrm{par} : V \to V$. *Let $p \in \mathbb{Z}_{\geq 0}$. Let $s \in \mathbb{Z}_{\geq 1}$. Let $Q \subseteq V \times \mathbb{Z}_{\geq 0} \times [s]$ be a set of m-tuples.* ColorSameLen(par, Q, p) *(Algorithm 26) can be implemented in the $(0, \delta)$-MPC model for any constant $\delta \in (0, 1)$ in $O(p)$ parallel time.*

Proof. $g^{(0)}(v)$ and $\mathrm{col}^{(0)}(v)$ can be computed in $O(N)$ total space and $O(1)$ parallel time using **Multiple queries** and sorting (Theorem 2.3.1). In the i-th iterations, we only need to store $\mathrm{col}^{(i-1)}, \mathrm{col}^{(i)}, g^{(i-1)}$ and $g^{(i)}$. Thus, the total space needed is $O(m + n)$. It is easy to verify that each iteration can be implemented in $O(1)$ parallel time by using **Multiple queries** and sorting (Theorem 2.3.1). □

Lemma 5.7.3. *Consider an n-vertex rooted tree represented by a set of parent pointers* $\mathrm{par} : V \to V$. *Let $s \in \mathbb{Z}_{\geq 1}$. Let $Q \subseteq V \times \mathbb{Z}_{\geq 0} \times [s]$ be a set of m tuples. Then* ColorDifLen(par, Q) *(Algorithm 27) can be implemented in (γ, δ)-MPC model for any constant $\delta \in (0, 1)$ and any $\gamma \geq 0$ satisfying $(m + n)^{1+\gamma} \geq m + n \log \mathrm{dep}(\mathrm{par})$. The parallel time needed is $O(\log(\mathrm{dep}(\mathrm{par})))$.*

Proof. According to Lemma 4.4.6, line 2 can be implemented in the MPC model in $O(n \log(\mathrm{dep}(\mathrm{par})))$ total space and $O(\log(\mathrm{dep}(\mathrm{par})))$ parallel time. Line 4 can be implemented using sorting (Theorem 2.3.1) in $O(1)$ parallel time and $O(m + n)$ total space. Then the algorithm has t iterations. For the iteration with i, we need to store $Q'_{i-1}, Q'_i, Q_{i-1}, Q_i, Q_{i+1}, \cdots, Q_t$ in the space. Notice that each Q_k for $k \geq i - 1$ has size $O(n)$, and $|Q'_{i-1}|, |Q'_i| \leq m + n$ according to Lemma 5.6.7. Thus, the total space needed is at most $nt + m = O(n \log(\mathrm{dep}(\mathrm{par})) + m)$. Each step in the iteration can be implemented by sorting (Theorem 2.3.1) and **Multiple queries**. Line 11 can be implemented simultaneously for every i (see **Multiple tasks**). According to Lemma 5.7.2, the total space needed is $O(nt) = O(n \log(\mathrm{dep}(\mathrm{par})))$. Thus, the overall total space needed is at most $O(n \log(\mathrm{dep}(\mathrm{par})) + m)$ and the parallel time needed is $O(\log(\mathrm{dep}(\mathrm{par})))$. □

Lemma 5.7.4. *Consider an n-vertex rooted spanning tree represented by a set of parent pointers* $\mathrm{par} : V \to V$ *and a set of m tuples* $Q \subseteq V \times \mathbb{Z}_{\geq 0}$. TARANCESTOR$(\mathrm{par}, Q)$ *(Algorithm 28) can be implemented in the* $(0, \delta)$*-MPC model for any constant* $\delta \in (0, 1)$ *and* $O(\log(\mathrm{dep}(\mathrm{par})))$ *parallel time.*

Proof. According to lemma 4.4.2, the depth of each vertex in par can be computed in $O(n)$ space and $O(\log(\mathrm{dep}(\mathrm{par})))$ parallel time. Each remaining step can be implemented in $O(n + m)$ total space and $O(1)$ parallel time by **Multiple queries**. Since the algorithm has $t = O(\log(\mathrm{dep}(\mathrm{par})))$ iterations, the overall total space needed is $O(n + m)$ and the overall parallel time is $O(t) = O(\log(\mathrm{dep}(\mathrm{par})))$. □

Lemma 5.7.5. *Consider an n-vertex rooted tree represented by a set of parent pointers* $\mathrm{par} : V \to V$*. Let* $s \in \mathbb{Z}_{\geq 1}$*. Let* $Q \subseteq V \times \mathbb{Z}_{\geq 0} \times [s]$ *be a set of m tuples.* COLORING(par, Q) *(Algorithm 29) can be implemented in the* $(0, \delta)$*-MPC model and* $O(\log(\mathrm{dep}(\mathrm{par})))$ *parallel time.*

Proof. Notice that the steps before line 6 are also implemented as the first several steps of Algorithm 14, according to Lemma 5.3.3, the total space needed is at most $O(n+m)$ and the parallel time is at most $O(\log(\mathrm{dep}(\mathrm{par})))$. We can handle all tuples $(u, l, c) \in Q$ in the loop of line 10 simultaneously. Line 12 can be implemented in $O(n+m)$ space and $O(10 \cdot t) = O(\log(\mathrm{dep}(\mathrm{par})))$ parallel time. Line 14 can be implemented in $O(2t) = O(\log(\mathrm{dep}(\mathrm{par})))$ and $O(n + m)$ total space. According to Lemma 5.7.4, line 16 can be implemented in $O(\log(\mathrm{dep}(\mathrm{par})))$ parallel time and $O(n + m)$ total space. Line 18 and line 19 can be implemented in $O(2t + l - h) = O(t) = O(\log(\mathrm{dep}(\mathrm{par})))$ parallel time and $O(n + m)$ total space. Notice that the size of Q' is at most $|Q|$ and $\mathrm{dep}(\mathrm{par}') \leq \mathrm{dep}(\mathrm{par})$. According to Lemma 5.2.7, $|V'| \leq |V|/\log(\mathrm{dep}(p))$. According to Lemma 5.7.3, line 22 can be implemented in the MPC model in $O(\log(\mathrm{dep}(\mathrm{par})))$ parallel time and $O(|V'| \log(\mathrm{dep}(\mathrm{par}')) + |Q'|) = O(n+m)$ space. Line 23 can be implemented in $O(n+m)$ space and $O(t) = O(\log(\mathrm{dep}(\mathrm{par})))$ parallel time. Thus, the overall space needed is $O(n + m)$ and the parallel time is $O(\log(\mathrm{dep}(\mathrm{par})))$. □

Open ear decomposition in MPC

Now we are able to describe the implementation of Algorithm 30 in the MPC model. Consider an n-vertex m-edge biconnected graph G with diameter D and bi-diameter D'.

Theorem 5.7.6 (Open ear decomposition in MPC)**.** *For any $\gamma \in [0,2]$ and any constant $\delta \in (0,1)$, there is a randomized (γ,δ)-MPC algorithm which outputs an open ear decomposition of G in $O\left(\log D \cdot \log^2 \frac{\log n}{\log(N^{1+\gamma}/n)} + \log D' \cdot \log \frac{\log n}{\log(N^{1+\gamma}/n)}\right)$ parallel time. The success probability is at least 0.94. If the algorithm fails, then it returns FAIL.*

Proof. Let $E_1, E_2, \cdots, E_s$ be the output of OPENEARDECOMP(G) (Algorithm 30). As shown by Lemma 5.6.12, $E_1, E_2, \cdots, E_s$ is an open ear decomposition of G.

We show how to implement OPENEARDECOMP(G) (Algorithm 30) in the MPC model. Firstly, the only reason that Algorithm 30 may fail is that the computation[2] of DFS sequence or the rooted spanning tree may fail. According to Theorem 5.3.9 and Theorem 4.4.12, if the DFS sequence algorithm or the spanning tree algorithm fails, it will output FAIL. Due to Theorem 4.4.12, the success probability of computing par$'$ is at least 0.98. Due to Lemma 5.7.5, the success probability of computing ORDERING(G, par) (Algorithm 25) is at least 0.96. Thus, the overall success probability is at least 0.94.

Let $\gamma' = (1+\gamma)\log_n(2N/(n^{1/(1+\gamma)}))$. According to Theorem 4.4.12, the computation of par can be done in (γ,δ)-MPC model in $O(\min(\log D \cdot \log \frac{1}{\gamma'}, \log n))$ parallel time and the depth of par is at most $D^{O(\log(1/\gamma'))}$. According to Lemma 5.7.1, ORDERING(G, par) (Algorithm 25) can be computed in (γ,δ)-MPC model in $O\left((\log D' + \log D \cdot \log 1/\gamma') \cdot \log \frac{\log n}{\log(N^{1+\gamma}/n)}\right)$ parallel time. According to Lemma 5.3.3, line 5 can be implemented in the $(0,\delta)$-MPC model and $O(D' \cdot \log(1/\gamma'))$ parallel time. According to Lemma 4.4.2, $\mathrm{dep}_{\mathrm{par}} : V \to \mathbb{Z}_{\geq 0}$ can be computed in $(0,\delta)$-MPC model and $O(D \cdot \log(1/\gamma'))$ parallel time. According to Lemma 5.7.5, col can be computed in $(0,\delta)$-MPC model and $O(D \cdot \log(1/\gamma'))$ parallel time.

Thus, algorithm can be implemented in the (γ,δ)-MPC model and the overall parallel time is

[2] Also include the computation in the subroutines of Algorithm 30.

$$O\left(\log D \cdot \log^2 \frac{\log n}{\log(N^{1+\gamma}/n)} + \log D' \cdot \log \frac{\log n}{\log(N^{1+\gamma}/n)}\right). \qquad \square$$

Chapter 6: Shortest Path and Uncapacitated Minimum Cost Flow

In this chapter, we show how to design efficient parallel algorithms for undirected shortest path and uncapacitated minimum cost flow problem. We will first introduce several new graph concepts such as subemulator and low hop emulator. We will show how to use truncated broadcasting and double-exponential speed problem size reduction techniques to construct subemulators and low hop emulators. Then we will show how to use them to design efficient algorithms for shortest path and uncapacitated minimum cost flow.

6.1 Overview of techniques

In this section, we give an overview of techniques that we use in our algorithms. Figure 6.1 sketches the dependencies between our techniques and the main algorithms.

6.1.1 Low hop emulator

We introduce a new notion — *low hop emulator*. A low hop emulator $G' = (V, E', w')$ of G is a sparse graph with $n \operatorname{poly}(\log n)$ edges satisfying two properties. First, the distance between every pair of vertices in G' is a $\operatorname{poly}(\log n)$ approximation to the distance in G. The second property is that G' has a low hop diameter, i.e., a shortest path between every pair of two vertices in G' only contains $O(\log \log n)$ number of hops (edges). A concept closely related to low hop emulator is hopset [34]. A hopset is a set of weighted shortcut edges such that for any two vertices s and t we can always find an approximate shortest path connecting them using small number of edges from the hopset and the original graph. Many hopset construction methods [34, 28, 89, 52, 37, 90, 91, 42, 38, 53] share some common features — they all choose a layer or multiple layers of leader vertices, and the hopset edges are some shortcut edges connecting to these leader vertices.

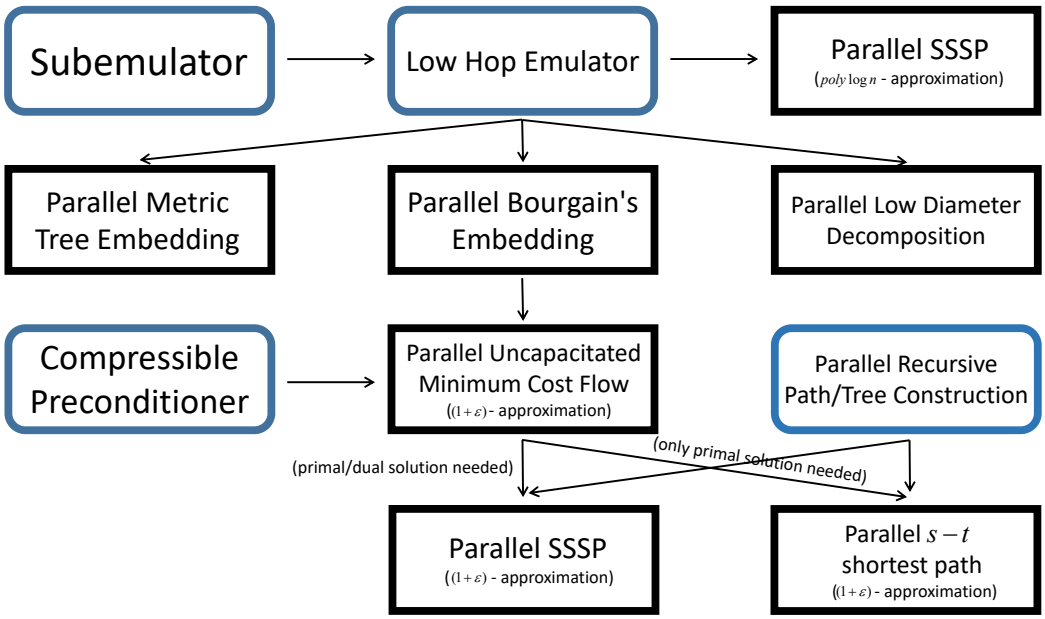

Figure 6.1: A summary of techniques and main algorithms. Blue rounded rectangles indicate new techniques.

However, when connecting shortcut edges to a layer of leader vertices, none of these algorithms can avoid processing information for all n vertices from the original graph, even though there may be a large fraction of vertices which are not connecting any this layer's leader vertex in the final hopset. Furthermore, each of these algorithms needs either $n \cdot \log^{\omega(1)} n$ work (sequential time) or $\log^{\omega(1)} n$ depth to process n vertices for constructing shortcut edges for some layers. To improve the efficiency of these algorithms, a natural question is: can we reduce the number of vertices needed to be processed when constructing the shortcut edges?

Subemulator: Motivated by the above question, we introduce a new concept called subemulator. For $\alpha \geq 1$ and an integer $b \geq 1$, we say $H = (V', E', w')$ is an (α, b)-subemulator of $G = (V, E, w)$ if 1) V' is a subset of V; 2) for any vertex v in G, at least one of the b-closest vertices of v is in V'; 3) for any two vertices u, v in H, $\text{dist}_H(u, v)$ α-approximates $\text{dist}_G(u, v)$. In addition, if we can assign each vertex $v \in V$ a leader $q(v) \in V'$ such that $q(v)$ is one of the b-closest vertices of v and

for any two vertices $u, v \in V$ it always satisfies

$$\text{dist}_G(q(u), q(v)) \leq \text{dist}_H(q(u), q(v)) \leq \text{dist}_G(q(u), u) + \beta \cdot \text{dist}_G(u, v) + \text{dist}_G(v, q(v)) \tag{6.1}$$

for some $\beta \geq 1$, we call H a strong (α, b, β)-subemulator of G. A subemulator H can be regarded as a sparsification of vertices of G. Two notions related to subemulators are vertex sparsifiers [92, 93] and distance-preserving minors [94]. The major difference between subemulators and vertex sparisfiers is that the vertex sparsifier approximately preserves flow/cut properties for the subset of vertices while the subemulator approximately preserves distances for the subset of vertices. Furthermore, both vertex sparsifiers and distance-preserving minors have given fixed vertex sets, whereas the vertex set of the subemulator is not given but should satisfy the condition 2) mentioned above, i.e., each vertex in G has a b-closest neighbor which is in the subemulator.

To construct a strong subemulator $H = (V', E', w')$, we need to construct both a vertex set V' and a edge set E'. For convenience, let us consider the case for $b \gg \log n$. Constructing V' is relatively easy. We can add each vertex of V to V' with probability $\Theta(\log(n)/b)$. By Chernoff bound, with high probability, each vertex has at least one of the b-closest vertices in V' and the size of V' is roughly $\widetilde{O}(n/b)$. For each vertex $v \in V$, it is natural to set the leader vertex $q(v)$ to be the vertex in V' which is the closest vertex to v. The challenge remaining is to construct the edge set E' such that condition 3) and Equation (6.1) can be satisfied. In our construction, we add two categories of edges to E':

1. For each edge $\{u, v\} \in E$, add an edge $\{q(u), q(v)\}$ with weight $\text{dist}_G(q(u), u) + w(u, v) + \text{dist}_G(v, q(v))$ to E'.

2. For each $v \in V$ and for each u which is a b-closest vertex of v, we add an edge $\{q(u), q(v)\}$ with weight $\text{dist}_G(q(u), u) + \text{dist}_G(u, v) + \text{dist}_G(q(v), v)$ to E'.

The first category of edges looks natural — for an edge $\{u, v\}$ of which two end points u, v are assigned to different leader vertices $q(u), q(v)$, we add a shortcut edge connecting those two leader vertices with a weight which is equal to the smallest length of the $q(u) - q(v)$ path crossing edge

$\{u, v\}$. However, if we only have the edges from the first category, it is not good enough to preserve the distances between leader vertices (see Section 7.4 for examples). To fix this, we add the second category of edges. We now sketch the analysis. It follows from our construction that each edge in H corresponds to a path in G. Thus, $\forall u', v' \in V'$, $\text{dist}_G(u', v') \leq \text{dist}_H(u', v')$. We only need to upper bound $\text{dist}_H(u', v')$. Let us fix a shortest path $u' = z_0 \to z_1 \to \cdots \to z_h = v'$ between u', v' in the original graph G. We want to construct a path in H with a short length. We use the following procedure to find some crucial vertices on the shortest path $z_0 \to \cdots \to z_h$:

1. $y_0 \leftarrow u', k \leftarrow 0$. Repeat the following two steps:
2. Let x_{k+1} be the last vertex on the path $z_0 \to \cdots \to z_h$ such that x_{k+1} is one of the b-closest vertices of y_k. If x_{k+1} is z_h, finish the procedure.
3. Set y_{k+1} to be the next vertex of x_{k+1} on the path $z_0 \to \cdots \to z_h$. $k \leftarrow k + 1$.

It is obvious that

$$\text{dist}_G(u', v') = \text{dist}_G(y_k, x_{k+1}) + \sum_{i=0}^{k-1} (\text{dist}_G(y_i, x_{i+1}) + w(x_{i+1}, y_{i+1})).$$

For $i = 0, 1, \cdots, k$, x_{i+1} is a b-closest vertex of y_i. Thus, there is an edge $\{q(y_i), q(x_{i+1})\}$ in H from the second category of the edges. For $i = 1, 2, \cdots, k$, y_i is adjacent to x_i. Thus, there is an edge $\{q(x_i), q(y_i)\}$ in H from the first category of the edges. Thus $u' = q(y_0) \to q(x_1) \to q(y_1) \to q(x_2) \to q(y_2) \to \cdots \to q(x_{k+1}) = v'$ is a valid path (see Figure 6.2) in H and the length is

$$\text{dist}_G(u', v') + 2 \cdot \sum_{i=1}^{k} (\text{dist}_G(x_i, q(x_i)) + \text{dist}_G(y_i, q(y_i))) .$$

By our choice of $q(\cdot)$, we have $\forall v \in V, \text{dist}_G(v, q(v)) = \text{dist}_G(v, V')$. So,

$$\forall i = 1, 2, \cdots, k, \text{dist}_G(x_i, q(x_i)) \leq \text{dist}_G(y_{i-1}, q(y_{i-1})) + \text{dist}_G(y_{i-1}, x_i).$$

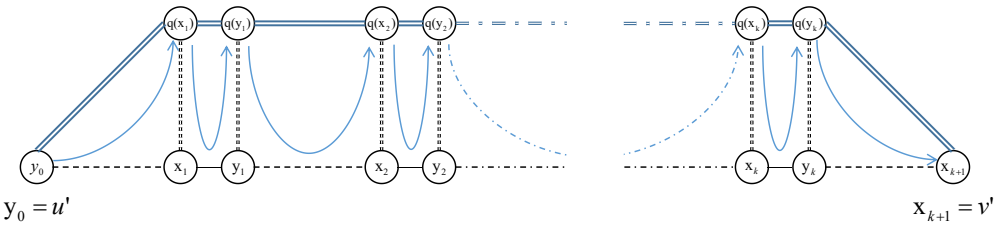

Figure 6.2: For $u', v' \in V'$ and a shortest path between u', v' in G, we can find a corresponding path between u', v' in the subemulator H. A single dashed line denotes a shortest path in G between y_{i-1} and x_i. A single solid line denotes an edge $\{x_i, y_i\}$ in G. A double dashed line denotes a shortest path in G between a vertex and its leader vertex. A double solid blue line denotes an edge in the subemulator H with a weight which is equal to the length of the path in G represented by the corresponding blue arc.

Since y_i is not a b-closest vertex of y_{i-1} but $q(y_{i-1})$ is a b-closest vertex of y_{i-1},

$$\forall i = 1, 2, \cdots, k, \mathrm{dist}_G(y_{i-1}, q(y_{i-1})) \leq \mathrm{dist}_G(y_{i-1}, x_i) + w(x_i, y_i).$$

Since $x_{k+1} \in V'$, we have $\mathrm{dist}_G(y_k, q(y_k)) \leq \mathrm{dist}_G(y_k, x_{k+1})$. Then we know $\sum_{i=1}^{k} \mathrm{dist}_G(x_i, q(x_i)) \leq 2 \cdot \mathrm{dist}_G(u', v')$ and $\sum_{i=1}^{k} \mathrm{dist}_G(y_i, q(y_i)) \leq \mathrm{dist}_G(u', v')$. Thus, we can conclude $\mathrm{dist}_H(u', v') \leq 8 \cdot \mathrm{dist}_G(u', v')$. We now argue that our construction of E' also satisfies Equation (6.1) with $\beta = 22$. There are two cases. The first case is that either u is a b-closest vertex of v or v is a b-closest vertex of u. In this case, E' contains an edge from the second category with weight $\mathrm{dist}_G(q(u), u) + \mathrm{dist}_G(u, v) + \mathrm{dist}_G(v, q(v))$ which implies Equation (6.1). The second case is that neither u is a b-closest vertex of v nor v is a b-closest vertex of u. In this case, we have

$$\begin{aligned}\mathrm{dist}_H(q(u), q(v)) &\leq 8\,\mathrm{dist}_G(q(u), q(v)) \leq 8(\mathrm{dist}_G(q(u), u) + \mathrm{dist}_G(u, v) + \mathrm{dist}_G(v, q(v))) \\ &\leq \mathrm{dist}_G(q(u), u) + \mathrm{dist}_G(v, q(v)) + 22\,\mathrm{dist}_G(u, v),\end{aligned}$$

where the last step follows from $\mathrm{dist}_G(u, q(u)), \mathrm{dist}_G(v, q(v)) \leq \mathrm{dist}_G(u, v)$.

The bottleneck of computing a subemulator is to obtain b-closest vertices for each vertex. We can use the truncated broadcasting technique (see Algorithm 1) to handle this in $\mathrm{poly}(\log n)$ parallel time using $\widetilde{O}(m + nb^2)$ total work (see Lemma 3.1.3). The output subemulator has $\widetilde{O}(n/b)$ vertices

and $O(m + nb)$ edges. As we can see, there is a trade-off between total work used and the number of vertices in the subemulator: if we can afford more work for the construction of the subemulator, fewer vertices appear in the subemulator.

Low hop emulator via subemulator: Now, we describe how to use strong subemulators to construct a low hop emulator. Consider a weighted undirected graph $G = (V, E, w)$. Suppose we obtain a sequence of subemulators $H_0 = (V_0, E_0, w_0), H_1 = (V_1, E_1, w_1), \cdots, H_t = (V_t, E_t, w_t)$ where $H_0 = G$ and $\forall i = 0, \cdots, t-1$, H_{i+1} is a strong $(8, b_i, 22)$-subemulator of H_i for some integer $b_i \geq 1$. We have $V = V_0 \supseteq V_1 \supseteq V_2 \supseteq \cdots \supseteq V_t$. For $v \in V_i$, let us denote $q_i(v) \in V_{i+1}$ as the corresponding assigned leader vertex of v in the subemulator H_{i+1} satisfying Equation (6.1). We add following three types of edges to the graph $G' = (V, E', w')$ and we will see that G' is a low hop emulator of G:

1. $\forall i = 0, \cdots, t-1$, $\forall v \in V_i$, add an edge $\{v, q_i(v)\}$ with weight $27^{t-i-1} \cdot \mathrm{dist}_{H_i}(v, q_i(v))$ to G'.
2. $\forall i = 0, \cdots, t$, for each edge $\{u, v\} \in E_i$, add an edge $\{u, v\}$ with weight $27^{t-i} \cdot w_i(u, v)$ to G'.
3. $\forall i = 0, \cdots, t$, $\forall v \in V_i$, add an edge $\{v, u\}$ with weight $27^{t-i} \cdot \mathrm{dist}_{H_i}(v, u)$ to G' for each u which is one of the b_i-closest vertices of v in H_i (define $b_t = |V_t|$).

Roughly speaking, we can imagine that G' is obtained from flattening a graph with $t + 1$ layers. Each layer corresponds to a subemulator. The lowest layer corresponds to the original graph G, and the highest layer corresponds to the last subemulator H_t. The first type of edges connect the vertices in the lower layer to the leader vertices in the higher layer. The second type of edges correspond to the subemulators on all layers. The third type of edges shortcut the close vertices from the same layer. Furthermore, the weights of the edges on the lower layers have larger penalty factor, i.e., the penalty factor of the edges on the layer i is 27^{t-i}.

By Equation (6.1) of strong subemulator, we can show that $\forall u, v \in V, \mathrm{dist}_G(u, v) \leq \mathrm{dist}_{G'}(u, v)$. Consider the first layer. By the second type edges, we know that $\forall u, v \in V, \mathrm{dist}_{G'}(u, v) \leq 27^t \mathrm{dist}_G(u, v)$. In particular, for $t = O(\log \log n)$, G' preserves the distances in G up to a $\mathrm{poly}(\log n)$ factor. Now

we want to show that $\forall u, v \in V$, there is always a shortest path connecting u, v in G' such that the number of hops (edges) of the path is at most $4t$. For convenience, we conceptually split each vertex of G' into vertices on different layers based on the construction of G'. Consider a shortest path $u = z_0 \to z_1 \to z_2 \to \cdots \to z_h = v$ using the smallest number of hops in G' with splitting vertices. By the constructions of three types of edges we know that $\forall j = 0, 1, \cdots, h-1$, z_j, z_{j+1} are either on the same layer or on the adjacent layers, and z_0, z_h are on the lowest layer which is corresponding to $H_0 = G$. We will claim two properties of the shortest path $z_0 \to \cdots \to z_h$. Suppose z_j, z_{j+1} are on the same layer corresponding to H_i. We claim that z_{j+2} cannot be on the same layer as z_j and z_{j+1}. Intuitively, this is because if z_{j+2} is on the same layer of z_j then there are two cases which both lead to contradictions: in the first case, z_{j+2} is close to z_j such that there is a third type edge connecting z_j, z_{j+2} which implies that z_{j+1} is redundant; in the second case, z_{j+2} is far away from z_j such that $\text{dist}_{H_i}(z_j, q_i(z_j)) + \text{dist}_{H_{i+1}}(q_i(z_j), q_i(z_{j+2})) + \text{dist}_{H_i}(q_i(z_{j+2}), z_{j+2})$ is a good approximation to $\text{dist}_{H_i}(z_j, z_{j+2})$, and due to a smaller penalty factor, the length of the path $z_j \to q_i(z_j) \to$ (shortest path) $\to q_i(z_{j+2}) \to z_{j+2}$ is smaller than the length of $z_j \to z_{j+1} \to z_{j+2}$. We claim another property of $z_0 \to \cdots \to z_h$ as the following. If the layer of z_{j+1} is lower than the layer of z_j, the layer of any of $z_{j+2}, z_{j+3}, \cdots, z_h$ must be lower than the layer of z_j. At a high level, this is because of Equation (6.1) and the smaller penalty factor for higher layers: if we move from higher layer to lower layer then come back to the higher layer, it is always worse than we only move in the higher layers. Due to these two claims, the shortest path in G' should have the following shape: the path starts from the lowest layer, then keeps moving to the non-lower layers until reach some vertex, and finally keeps moving to the non-higher layers until reach the target. Furthermore, there are no three consecutive vertices on the path which are on the same layer. Hence we can conclude that the shortest path has number of hops at most $4t$. Based on above analysis, the shortest path in G' will never use the second type edges. Thus, in our final construction of G', we only need the first type and the third type of edges.

The size of G' is at most $\sum_{i=0}^{t} |V_i| \cdot b_i$. The bottleneck of the construction of G' is to compute the third type edges. This can be done by truncated broadcasting (Algorithm 1) in $t \cdot \text{poly}(\log n)$ paral-

lel time using $\sum_{i=0}^{t} \left(|E_i| + |V_i| \cdot b_i^2\right) \cdot \text{poly}(\log n)$ total work (Lemma 3.1.3). The problem remaining is to determine the sequence of b_i. As we discussed previously, we are able to use $\text{poly}(\log |V_i|)$ parallel time and $\widetilde{O}(|E_i| + |V_i| b_i)$ total work to construct a subemulator H_{i+1} with $\widetilde{O}(|V_i|/b_i)$ vertices and $O(|E_i| + |V_i| b_i)$ edges. By double-exponential problem size reduction technique (see Section 3.2), we can make b_i grow double exponentially fast in this situation. More precisely, if we set $b_0 \leftarrow \text{poly}(\log n)$, $b_{i+1} \leftarrow b_i^{1.25}$, and $t \leftarrow O(\log \log n)$, then in this case, the result low hop emulator can be computed in $\text{poly}(\log n)$ parallel time and $\widetilde{O}(m+n)$ total work. Furthermore, the size of the result low hop emulator is $\widetilde{O}(n)$, the approximation ratio is $\text{poly}(\log n)$, and the hop diameter is $O(\log \log n)$.

Applications of low hop emulator: We can build a useful oracle based on a low hop emulator: given a query subset S of vertices, the oracle can output a $\text{poly}(\log n)$ approximations to $\text{dist}_G(v, S)$ for all $v \in V$. Furthermore, the output approximate distances always satisfy triangle inequality. To implement the such oracle, we preprocess an $\widetilde{O}(n)$ size low hop emulator G' with $\text{poly}(\log n)$ approximation ratio and $O(\log \log n)$ hop diameter in $\text{poly}(\log n)$ parallel time using $\widetilde{O}(m+n)$ work. For each oracle query, we can just run Bellman-Ford on G' with source S. The work needed for each Bellman-Ford iteration is at most $\widetilde{O}(n)$. Since the hop diameter is $O(\log \log n)$, the number of iterations needed is $O(\log \log n)$. Therefore, each query can be handled in $\text{poly}(\log n)$ parallel time and $\widetilde{O}(n)$ total work. The triangle inequality is always satisfied since the output approximate distances are exact distances in the graph G'. Several parallel applications such as Bourgain's embedding, metric tree embedding and low diameter decomposition directly follow the oracle.

6.1.2 Minimum cost flow and shortest path

Uncapacitated minimum cost flow: At a high level, our uncapacitated minimum cost flow algorithm is based on Sherman's framework [46]. Sherman's algorithm has several recursive iterations. It first uses the multiplicative weights update method [95] to find a flow which almost satisfies the demands and has nearly optimal cost. If the unsatisfied parts of demands are sufficiently small, it

routes them naively to make the flow truly feasible without increasing the cost by too much. Otherwise, it updates the demands to be the unsatisfied parts of the original demands and recursively routes the new demands. [46] shows that if the problem is well conditioned, then the final solution can be computed by the above process efficiently. However, most of the time the natural form of the uncapacitated minimum cost flow problem is not well-conditioned. Thus, a preconditioner, i.e., a linear operator $P \in \mathbb{R}^{r\times n}$ applied to the flow constraints, is needed to make the problem well-condtitioned. Consider a given graph $G = (V, E, w)$. Sherman shows that if for any valid demands $b \in \mathbb{R}^n$ we always have

$$\mathrm{OPT}(b) \leq \|Pb\|_1 \leq \gamma \cdot \mathrm{OPT}(b),$$

then P can make the condition number of the flow problem on G be upper bounded by γ, where $\mathrm{OPT}(b)$ denotes the optimal cost of the flow on G satisfying the demands b. Sherman gives a method to construct such P. However, to have a smaller approximation ratio γ, the time of computing matrix-vector multiplication with P must increase such that the running time of the multiplicative weights update step increases. To balance the trade-off, Sherman constructs P with $\gamma = 2^{O(\sqrt{\log n})}$ approximation ratio and $\mathrm{nnz}(x) \cdot 2^{O(\sqrt{\log n})}$ time for matrix-vector multiplication $P \cdot x$, where $\mathrm{nnz}(x)$ denotes the number of non-zero entries of x. Thus, its final running time is $m \cdot 2^{O(\sqrt{\log n})}$. To design a parallel minimum cost flow algorithm using $\mathrm{poly}(\log n)$ parallel time and $m\,\mathrm{poly}(\log n)$ work, we cannot avoid improving the sequential running time of minimum cost flow to $m\,\mathrm{poly}(\log n)$ time in sequential setting. By above discussion, a natural way is to find a linear transformation P which can embed the uncapacitated minimum cost flow into ℓ_1 with $\mathrm{poly}(\log n)$ approximation ratio and the running time for matrix-vector multiplication $P \cdot x$ needs to be $\mathrm{nnz}(x) \cdot \mathrm{poly}(\log n)$. Next, we will introduce how to construct such embedding P.

First, we compute a mapping φ which embeds the vertices into ℓ_1^d for $d = O(\log^2 n)$ such that $\forall u, v \in V$, $\|\varphi(u) - \varphi(v)\|_1$ is a $\mathrm{poly}(\log n)$ approximation to $\mathrm{dist}_G(u, v)$. This step can be done by Bourgain's embedding. The parallel version of Bourgain's embedding is one of the applications

of low hop emulator as we mentioned previously. Then we can reduce the minimum cost flow problem to the geometric transportation problem. The geometric transportation problem is also called Earth Mover's Distance (EMD) problem. Specifically, it is the following minimization problem:

$$\min_{\pi: V\times V\to\mathbb{R}_{\geq 0}} \sum_{(u,v)\in V\times V} \pi(u,v)\cdot \|\varphi(u)-\varphi(v)\|_1$$
$$s.t.\ \forall u\in V, \sum_{v\in V}\pi(u,v)-\sum_{v\in V}\pi(v,u)=b_u.$$

We denote $\mathrm{OPT}_{\mathrm{EMD}}(b)$ as the optimal cost of the above EMD problem. It is easy to see that $\mathrm{OPT}_{\mathrm{EMD}}(b)$ is a $\mathrm{poly}(\log n)$ approximation to $\mathrm{OPT}(b)$. Therefore, it suffices to construct P such that for any valid demand vector $b\in\mathbb{R}^n$,

$$\mathrm{OPT}_{\mathrm{EMD}}(b)\leq \|Pb\|_1 \leq \mathrm{poly}(\log n)\cdot \mathrm{OPT}_{\mathrm{EMD}}(b).$$

One known embedding of EMD into ℓ_1 is based on randomly shifted grids [96]. We can without loss of generality assume that the coordinates of $\varphi(v)$ are integers in $\{1,\cdots,\Delta\}$ for some Δ which is a power of 2 and upper bounded by $\mathrm{poly}(n)$. We create $1+\log\Delta$ levels of cells. We number each level from 0 to $\log\Delta$. Each cell in level $\log\Delta$ has side length Δ. Each cell in level $i+1$ is partitioned into 2^d equal size cells in level i and thus each cell in level i has side length 2^i. Therefore each cell in level 0 can contain at most one point $\varphi(v)$ for $v\in V$. According to [96], for any valid demand vector $b\in\mathbb{R}^n$,

$$\mathop{\mathbf{E}}_{\tau\sim\{0,1,\cdots,\Delta-1\}}\left[\sum_{i=0}^{\log\Delta}\sum_{C:\text{ a cell in level } i} 2^i\cdot\left|\sum_{v\in V:\varphi(v)+\tau\cdot\mathbf{1}_d \text{ is in the cell } C} b_v\right|\right] \tag{6.2}$$

is always a $\mathrm{poly}(\log n)$ approximation to $\mathrm{OPT}_{\mathrm{EMD}}(b)$, where τ is drawn uniformly at random from $\{0,1,\cdots,\Delta-1\}$, and $\varphi(v)+\tau\cdot\mathbf{1}_d$ is the point obtained after shifting each coordinate of $\varphi(v)$ by τ.

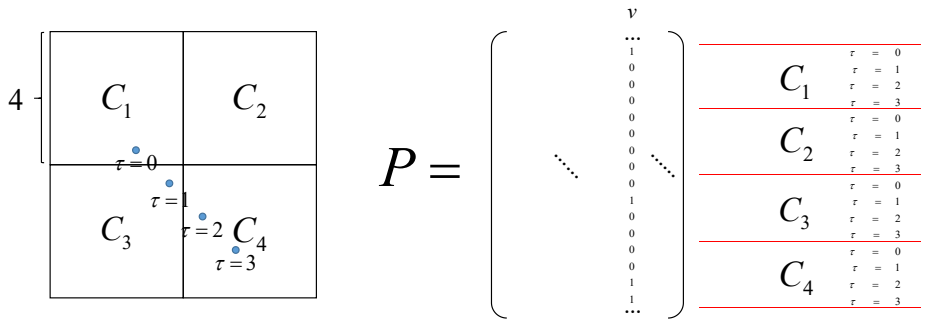

Figure 6.3: Consider cells C_1, C_2, C_3, C_4 shown above with side length 4. Blue dots denote the positions of $\varphi(v) + \tau \cdot \mathbf{1}_d$ for some vertex v and $\tau = 0, 1, 2, 3$. The entries of P in the column corresponding to v and in the rows corresponding to (C, τ) for $C = C_1, C_2, C_3, C_4$ and $\tau = 0, 1, 2, 3$ are shown on the right.

Since each cell in level i has side length 2^i, Equation (6.2) is equal to

$$\sum_{i=0}^{\log \Delta} \frac{1}{2^i} \sum_{\tau=0}^{2^i-1} \sum_{C\text{: a cell in level } i} 2^i \cdot \left| \sum_{v \in V : \varphi(v)+\tau\cdot\mathbf{1}_d \text{ is in the cell } C} b_v \right|$$
$$= \sum_{i=0}^{\log \Delta} \sum_{C\text{: a cell in level } i} \sum_{\tau=0}^{2^i-1} \left| \sum_{v \in V : \varphi(v)+\tau\cdot\mathbf{1}_d \text{ is in the cell } C} b_v \right|. \tag{6.3}$$

Equation (6.3) can be written in the from of $\|Pb\|_1$ where each row of P corresponds to a cell C and a shift value τ, and each column of P corresponds to a vertex v. Figure 6.3 shows how does P look like: for a entry $P_{i,j}$ corresponding to a cell C, a shift value τ and a vertex v, we have $P_{i,j} = 1$ if the point $\varphi(v) + \tau \cdot \mathbf{1}_d$ is in the cell C and $P_{i,j} = 0$ otherwise. Therefore, P can be used to precondition the minimum cost flow problem on G with condition number at most $\text{poly}(\log n)$. However, such matrix P is dense and have $\text{poly}(n)$ number of rows. It is impossible to naively write down the whole matrix. Fortunately, we will show that P has a good structure and we can write down a compressed representation of P. Consider a cell C in level i and a vertex v. If there exists $\tau \in \{0, 1, \cdots, 2^i - 1\}$ such that $\varphi(v) + \tau \cdot \mathbf{1}_d$ is in the cell C, then there must exist τ_1, τ_2 such that $\varphi(v) + \tau \cdot \mathbf{1}_d$ is in the cell C if and only if $\tau \in \{\tau_1, \tau_1 + 1, \cdots, \tau_2\}$. In other words, the shift values τ that can make $\varphi(v) + \tau \cdot \mathbf{1}_d$ be in C are consecutive. Another important property that we can show is that the number of cells in level i that can contain at least one of the shifted points $\varphi(v), \varphi(v) + \mathbf{1}_d, \varphi(v) + 2 \cdot \mathbf{1}_d, \cdots, \varphi(v) + (2^i - 1) \cdot \mathbf{1}_d$ is at most $d + 1$. Now consider a column of

P corresponding to some vertex v. The entries with value 1 in this column should be in several consecutive segments. The number of such segments is at most $(d+1)\cdot(1+\log\Delta) \le \text{poly}(\log n)$. Thus, for each column of P, we can just store the beginning and the ending positions of these segments. The whole matrix P can be represented by $n\,\text{poly}(\log n)$ segments. The only problem remaining is to use this compressed representation to do matrix-vector multiplication. Suppose we want to compute $y = P \cdot x$ for some $x \in \mathbb{R}^n$. It is equivalent to the following procedure:

1. Initialize y to be an all-zero vector.
2. For each column i and for each segment $[l, r]$ in column i, increase all $y_l, y_{l+1}, \cdots, y_r$ by x_i.

We can reduce the above procedure to the following one:

1. Initialize z to be an all-zero vector.
2. For each column i and for each segment $[l, r]$ in column i, increase z_l by x_i and increase z_{r+1} by $-x_i$.
3. Compute $y_j \leftarrow \sum_{k=1}^{j} z_k$.

In the above procedure, we only need to compute a prefix sum for z. Since each column of P has at most $\text{poly}(\log n)$ segments, the total number of segments involved is at most $\text{nnz}(x)\cdot\text{poly}(\log n)$. The total running time is $\widetilde{O}(\text{nnz}(x)\cdot\text{poly}(\log n))$. Notice that even though y has a large dimension, it can be decomposed into $\widetilde{O}(\text{nnz}(x)\cdot\text{poly}(\log n))$ segments where the entries of each segment have the same value. Thus, we just store the beginning and the ending positions of each segment of y.

Each step of computing the compressed representation can be implemented in $\text{poly}(\log n)$ parallel time and each step of computing the matrix-vector multiplication can also be implemented in $\text{poly}(\log n)$ parallel time. We obtained a desired preconditioner. By plugging this preconditioner into Sherman's framework, we can obtain a parallel $(1+\epsilon)$-approximate uncapacitated minimum cost flow algorithm with $\text{poly}(\log n)$ depth and $\epsilon^{-2}m\cdot\text{poly}(\log n)$ work.

Parallel $(1+\epsilon)$-approximate $s-t$ shortest path and single source shortest paths: $s-t$ Shortest path is closely related to uncapacitated minimum cost flow. If we set demand $b_s = 1, b_t = -1$ and $b_v = 0$ for $v \neq s, t \in V$, then the optimal cost of the flow is exactly $\text{dist}_G(s,t)$. Thus, computing a $(1+\epsilon)$-approximation to $\text{dist}_G(s,t)$ can be achieved by our flow algorithm. However, the flow algorithm can only output a flow but not a path. We need more effort to find a path from s to t with length at most $(1+\epsilon) \cdot \text{dist}_G(s,t)$. As mentioned by [43], if the $(1+\epsilon)$-approximate flow does not contain any cycles, then for each vertex $v \neq t$ we can choose an out edge with probability proportional to the magnitude of its out flow, and the expected length of the path found from s to t is exactly the cost of the flow which is $(1+\epsilon) \cdot \text{dist}_G(s,t)$. Unfortunately, the flow outputted by our flow algorithm may create cycles. If we randomly choose an out edge for each vertex $v \neq t$ with probability proportional to the magnitude of the out flow, we may stuck in some cycle and may not find a path from s to t. To handle cycles, we propose the following procedure to find a path from s to t.

1. If the graph only has constant number of vertices, find the shortest path from s to t directly.
2. Otherwise, compute the $(1+\epsilon')$-approximate minimum cost flow from s to t for $\epsilon' = \Theta(\epsilon/\log n)$.
3. For each vertex except t, choose an out edge with probability proportional to its out flow.
4. Consider the graph with $n-1$ chosen edges. Each connected component in the graph is either a tree or a tree plus an edge. A component is a tree if and only if t is in the component. For each component, we compute a spanning tree. If the component contains t, we set t as the root of the spanning tree. Otherwise, we set an arbitrary end point of the non-tree edge as the root of the spanning tree.
5. Construct a new graph of which vertices are roots of spanning trees. For each edge $\{u,v\}$ in the original graph, we add an edge connecting the root of u and the root of v with weight

$$\text{(distance from } u \text{ to the root of } u \text{ on the spanning tree)} + w(u,v)$$

+(distance from v to the root of u on the spanning tree).

6. Recursively find a $(1+\epsilon')$-approximate shortest path from the root of s to t in the new graph. Recover a path in the original graph from the path in the new graph.

In the above procedure, only $1/2$ vertices can be root vertices. Thus, the procedure can recurse at most $\log n$ times which implies that the parallel time of the algorithm is at most $\text{poly}(\log n)$ and the total work is still $\sim m\,\text{poly}(n)$. Now analyze the correctness. It is easy to see that each edge in the new graph corresponds to a path between two root vertices in the original graph. Thus a path from the root of s to t in the new graph corresponds to an $s-t$ path in the original graph. We only need to show that the distance between the root of s and t in the new graph can not be much larger than the distance between s and t in the original graph. To prove this, we show that if we do a random walk starting from s and for each step we choose the next vertex with probability proportional to the out flow, the expected length of the random walk to reach t is exactly the cost of the flow. By coupling argument, we can prove that the expected length of the distance between the root of s and t in the new graph is at most $(1+O(\epsilon'))\cdot$ (the cost of the flow). Thus, the expected length of the final $s-t$ path is at most $(1+O(\epsilon'))^{\log n}\cdot \text{dist}_G(s,t) \le (1+\epsilon)\cdot \text{dist}_G(s,t)$.

The single source shortest path (SSSP) problem is a more general problem. Given a source vertex $s \in V$, we want to approximate $\text{dist}_G(s,u)$ for every vertex $u \in V$ simultaneously. Consider a special case of the uncapacitated minimum cost flow problem. If we set demand $b_u \ge 0$ for all $u \ne s \in V$ and $b_s = -\sum_{u\ne s\in V} b_u$, then the optimal cost of the flow is exactly $\sum_{u\in V\setminus\{s\}} b_u \cdot \text{dist}_G(s,u)$. Thus, the flow routes on the shortest path tree is the optimal flow. We extend our recursive path construction method to recursively compute a tree T such that $\sum_{u\in V\setminus\{s\}} b_u \cdot \text{dist}_T(s,u)$ is a $(1+\epsilon)$-approximation to the optimal cost. Notice that the dual solution of the uncapacitated minimum cost flow can also be obtained by Sherman's algorithm [46]. By plugging these two subroutines into the framework proposed by [43], we are able to compute approximate single source shortest paths.

6.2 Low hop emulator

Given a weighted undirected graph G, we give a new construction of the graph emulator of G. For any two vertices in our constructed emulator, there is always a shortest path with small number of hops. Furthermore, our construction can be implemented in parallel efficiently.

6.2.1 Subemulator

In this section, we introduce *subemulator*. Later, we will show how to use subemulator to construct an emulator with low hop diameter.

Definition 6.2.1 (Subemulator)**.** *Consider two connected undirected weighted graphs $G = (V, E, w)$ and $H = (V', E', w')$. For $b \in [|V|]$ and $\alpha \geq 1$, if H satisfies*

1. $V' \subseteq V$,
2. $\forall v \in V, \mathrm{Ball}_{G,b}(v) \cap V' \neq \emptyset$,
3. $\forall u, v \in V', \mathrm{dist}_G(u, v) \leq \mathrm{dist}_H(u, v) \leq \alpha \cdot \mathrm{dist}_G(u, v)$,

then H is an (α, b)-subemulator of G. Furthermore, if there is a mapping $q : V \to V'$ which satisfies $\forall v \in V, q(v) \in \mathrm{Ball}_{G,b}(v)$ and

$$\forall u, v \in V, \mathrm{dist}_H(q(u), q(v)) \leq \mathrm{dist}_G(u, q(u)) + \mathrm{dist}_G(v, q(v)) + \beta \cdot \mathrm{dist}_G(u, v)$$

for some $\beta \geq 1$, then H is a strong (α, b, β)-subemulator of G, $q(\cdot)$ is called a leader mapping, and $q(v)$ is the leader of v.

Next, we will show how to construct a strong subemulator. Algorithm 31 gives a construction of the vertices of the subemulator. Algorithm 32 gives a construction of the edges of the subemulator.

The next lemma shows the correctness of the construction of the vertices of the subemulator. It also gives an upper bound of the number of vertices in the subemulator.

Algorithm 31 Construction of the Vertices of the Subemulator

1: **procedure** SAMPLES($G = (V, E, w), b \in [|V|]$)
2: Output: V'
3: Initialize $S, V' \leftarrow \emptyset$, $n \leftarrow |V|$
4: For $v \in V$, add v into S with probability $\min(50\log(n)/b, 1/2)$.
5: For $v \in V$, if either $v \in S$ or $\text{Ball}_{G,b}(v) \cap S = \emptyset$, $V' \leftarrow V' \cup \{v\}$.
6: Return V'.
7: **end procedure**

Algorithm 32 Construction of the Edges of the Subemulator

1: **procedure** CONNECTS($G = (V, E, w), V' \subseteq V, b \in [|V|]$) ▷ V', b satisfies $\forall v \in V, \text{Ball}_{G,b}(v) \cap V' \neq \emptyset$.
2: Output: $H = (V', E', w')$, $q : V \to V'$
3: For $v \in V$, $q(v) \leftarrow \arg\min_{u \in \text{Ball}_{G,b}(v) \cap V'} \text{dist}_G(u, v)$. ▷ Choose an arbitrary u if there is a tie.
4: Initialize $E' = \emptyset$.
5: For $\{u, v\} \in E$, $E' \leftarrow E' \cup \{\{q(u), q(v)\}\}$.
6: For $v \in V, u \in \text{Ball}_{G,b}(v)$, $E' \leftarrow E' \cup \{\{q(u), q(v)\}\}$.
7: For $e' \in E'$, initialize $w'(e') \leftarrow \infty$.
8: For $\{u, v\} \in E$, consider $e' = \{q(u), q(v)\} \in E'$,

$$w'(e') \leftarrow \min(w'(e'), \text{dist}_G(q(u), u) + w(u, v) + \text{dist}_G(v, q(v))).$$

9: For $v \in V, u \in \text{Ball}_{G,b}(v)$, consider $e' = \{q(u), q(v)\}$,

$$w'(e') \leftarrow \min(w'(e'), \text{dist}_G(q(u), u) + \text{dist}_G(u, v) + \text{dist}_G(v, q(v))).$$

10: Return $H = (V', E', w')$ and $q : V \to V'$.
11: **end procedure**

Lemma 6.2.2 (Construction of the vertices)**.** *Consider a connected n-vertex m-edge undirected weighted graph $G = (V, E, w)$ and a parameter $b \in [n]$.* SAMPLES(G, b) *(Algorithm 31) will output $V' \subseteq V$ such that $\forall v \in V, \text{Ball}_{G,b}(v) \cap V' \neq \emptyset$. Furthermore, $\mathbf{E}[|V'|] \leq \frac{3}{2} \cdot \min(50\log(n)/b, 1/2) \cdot n$.*

Proof. The correctness follows from line 5 of Algorithm 31, i.e., $\forall v \in V, \text{Ball}_{G,b}(v) \cap V' \neq \emptyset$.

Let us consider the expected size of V'. We have

$$\mathbf{E}[|V'|] = \mathbf{E}[|S|] + \mathbf{E}[|\{v \in V \mid \text{Ball}_{G,b}(v) \cap S = \emptyset\}|].$$

Consider the case when $1/2 < 50\log(n)/b$. We have $\mathbf{E}[|S|] = 1/2 \cdot |V|$. Notice that $\text{Ball}_{G,b}(v) \cap S = \emptyset$ implies that neither $v \in S$ nor $u \in S$, where $u \neq v$ is an arbitrary vertex in $\text{Ball}_{G,b}(v)$. Thus,

we have $\Pr[\mathrm{Ball}_{G,b}(v) \cap S = \emptyset] \leq 1/4$, and it implies that $\mathbf{E}[|V'|] \leq 1/2 \cdot |V| + 1/4 \cdot |V| = 3/4 \cdot n$.

Consider the case when $1/2 > 50\log(n)/b$. We have $\mathbf{E}[|S|] = 50\log(n)/b \cdot |V|$. Since $\mathbf{E}[|\mathrm{Ball}_{G,b}(v) \cap S|] = b \cdot 50\log(n)/b = 50\log(n)$, by Chernoff bound, we have

$$\Pr[\mathrm{Ball}_{G,b}(v) \cap S = \emptyset] \leq \Pr[|\mathrm{Ball}_{G,b}(v) \cap S| \leq 25\log(n)] \leq e^{-50\log(n)/8} \leq 1/n^2.$$

By union bound,

$$\Pr[\exists v, \mathrm{Ball}_{G,b}(v) \cap S = \emptyset] = 1/n.$$

Thus, $\mathbf{E}[|\{v \in V \mid \mathrm{Ball}_{G,b}(v) \cap S = \emptyset\}|] \leq 1/n \cdot n = 1$. We can conclude that $\mathbf{E}[|V'|] \leq 50\log(n)/b \cdot n + 1 \leq 75\log(n)/b \cdot n$. □

The next lemma shows the correctness of the construction of the edges of the subemulator. It also gives an upper bound of the number of edges in the subemulator.

Lemma 6.2.3 (Construction of the edges)**.** *Consider a connected n-vertex m-edge undirected weighted graph $G = (V, E, w)$, a vertex set $V' \subseteq V$ and a parameter $b \in [n]$. If $\forall v \in V, \mathrm{Ball}_{G,b}(v) \cap V' \neq \emptyset$, then the output graph $H = (V', E', w')$ of* Connects(G, V', b) *(Algorithm 32) will be a strong $(8, b, 22)$-subemulator (Definition 6.2.1) of G, and the output $q : V \to V'$ is a leader mapping. Furthermore, $|E'| \leq m + nb$.*

Proof. Firstly, let us consider the size of $|E'|$. The number of edges added to E' by line 5 of Algorithm 32 is at most m. By the definition of $\mathrm{Ball}_{G,b}(v)$, we have $|\mathrm{Ball}_{G,b}(v)| \leq b$. The number of edges added to E' by line 6 is at most $n \cdot b$. Thus, we can conclude $|E'| \leq m + nb$.

In the following, we will show that H is actually a good subemulator of G. The first two properties of Definition 6.2.1 are automatically satisfied by the guarantees of the input V', b. Let us prove the remaining properties.

Consider two arbitrary vertices $u, v \in V'$. Let $p = (u = x_0, x_1, \cdots, x_h = v)$ be an arbitrary shortest path between u, v in the graph H. Then $\mathrm{dist}_H(u, v) = w'(p) = \sum_{i=1}^{h} w'(x_{i-1}, x_i)$. By

line 8 and line 9, $\forall i \in [h]$, there should be $y_i, z_i \in V$ with $q(y_i) = x_{i-1}, q(z_i) = x_i$ such that $w'(x_{i-1}, x_i) \geq \mathrm{dist}_G(x_{i-1}, y_i) + \mathrm{dist}_G(y_i, z_i) + \mathrm{dist}_G(z_i, x_i) \geq \mathrm{dist}_G(x_{i-1}, x_i)$. Then, $\mathrm{dist}_H(u, v) = \sum_{i=1}^{h} w'(x_{i-1}, x_i) \geq \sum_{i=1}^{h} \mathrm{dist}_G(x_{i-1}, x_i) \geq \mathrm{dist}_G(x_0, x_h) = \mathrm{dist}_G(u, v)$.

Next, we show how to upper bound $\mathrm{dist}_H(u, v)$. Consider two arbitrary vertices $u, v \in V'$. If $v \in \mathrm{Ball}_{G,b}(u)$, then by line 9,

$$\mathrm{dist}_H(u, v) \leq w'(u, v) \leq \mathrm{dist}_G(q(u), u) + \mathrm{dist}_G(u, v) + \mathrm{dist}_G(v, q(v)) = \mathrm{dist}_G(u, v),$$

where the last equality follows from $q(u) = u, q(v) = v$. Otherwise, we use the following procedure to find a sequence of vertices which are on the shortest path between u and v in the graph G.

1. $y_0 \leftarrow u, t \leftarrow 0$.
2. If $v \in \mathrm{Ball}_{G,b}(y_t)$, finish the procedure.
3. Otherwise, find an edge $\{x_{t+1}, y_{t+1}\} \in E$ on the shortest path between y_t and v in G such that $x_{t+1} \in \mathrm{Ball}_{G,b}(y_t), y_{t+1} \notin \mathrm{Ball}_{G,b}(y_t)$.
4. $t \leftarrow t + 1$. Go to step 2.

By the above procedure, it is easy to show that

$$\mathrm{dist}_G(u, v) = \mathrm{dist}_G(y_t, v) + \sum_{i=1}^{t} (\mathrm{dist}_G(y_{i-1}, x_i) + w(x_i, y_i)). \tag{6.4}$$

Claim 6.2.4. $\sum_{i=1}^{t} r_{G,b}(y_{i-1}) \leq \mathrm{dist}_G(u, v)$.

Proof. By our construction of x_i, y_i, we know that $\forall i \in [t]$, $y_i \notin \mathrm{Ball}_{G,b}(y_{i-1})$. Thus, $\forall i \in [t]$, $r_{G,b}(y_{i-1}) \leq \mathrm{dist}_G(y_{i-1}, y_i)$. We have

$$\sum_{i=1}^{t} r_{G,b}(y_{i-1}) \leq \sum_{i=1}^{t} \mathrm{dist}_G(y_{i-1}, y_i) = \mathrm{dist}_G(u, y_t) \leq \mathrm{dist}_G(u, v).$$

□

Claim 6.2.5. $\forall u, v \in V, |r_{G,b}(u) - r_{G,b}(v)| \leq \text{dist}_G(u, v)$.

Proof. Since $\text{Ball}_G(u, r_{G,b}(u)) \subseteq \text{Ball}_G(v, r_{G,b}(u)+\text{dist}_G(u, v))$, we have $|\text{Ball}_G(v, r_{G,b}(u)+\text{dist}_G(u, v))| \geq b$ which implies that $r_{G,b}(v) \leq r_{G,b}(u) + \text{dist}_G(u, v)$. Similarly, we have $r_{G,b}(u) \leq r_{G,b}(v) + \text{dist}_G(u, v)$. □

Claim 6.2.6. $\forall i \in [t]$, $w'(q(y_{i-1}), q(x_i)) \leq 2r_{G,b}(y_{i-1}) + 2\,\text{dist}_G(y_{i-1}, x_i)$.

Proof. Since $x_i \in \text{Ball}_{G,b}(y_{i-1})$, we have $w'(q(y_{i-1}), q(x_i)) \leq \text{dist}_G(q(y_{i-1}), y_{i-1}) + \text{dist}_G(y_{i-1}, x_i) + \text{dist}_G(x_i, q(x_i)) \leq r_{G,b}(y_{i-1}) + \text{dist}_G(y_{i-1}, x_i) + r_{G,b}(x_i)$ by line 9. Due to Claim 6.2.5, $r_{G,b}(x_i) \leq r_{G,b}(y_{i-1}) + \text{dist}_G(y_{i-1}, x_i)$. We can conclude $w'(q(y_{i-1}), q(x_i)) \leq 2r_{G,b}(y_{i-1}) + 2\,\text{dist}_G(y_{i-1}, x_i)$. □

Claim 6.2.7. $\forall i \in [t]$, $w'(q(x_i), q(y_i)) \leq 2r_{G,b}(y_{i-1}) + 2\,\text{dist}_G(y_{i-1}, x_i) + 2w(x_i, y_i)$.

Proof. Since $\{x_i, y_i\} \in E$, we have $w'(q(x_i), q(y_i)) \leq \text{dist}_G(q(x_i), x_i) + w(x_i, y_i) + \text{dist}_G(y_i, q(y_i)) \leq r_{G,b}(x_i) + w(x_i, y_i) + r_{G,b}(y_i)$ by line 8. By Claim 6.2.5, $r_{G,b}(x_i) \leq r_{G,b}(y_{i-1}) + \text{dist}_G(y_{i-1}, x_i)$ and $r_{G,b}(y_i) \leq r_{G,b}(y_{i-1})+\text{dist}_G(y_{i-1}, x_i)+w(x_i, y_i)$. We can conclude that $w'(q(x_i), q(y_i)) \leq 2r_{G,b}(y_{i-1})+ 2\,\text{dist}_G(y_{i-1}, x_i) + 2w(x_i, y_i)$. □

Claim 6.2.8. $w'(q(y_t), v) \leq 2\,\text{dist}_G(y_t, v)$.

Proof. By our procedure of finding x_i, y_i, we know that $v \in \text{Ball}_{G,b}(y_t)$. Notice that $q(v) = v \in V'$. By line 3, we know that $\text{dist}_G(y_t, q(y_t)) \leq \text{dist}_G(y_t, v)$. Then by line 9, $w'(q(y_t), v) \leq \text{dist}_G(q(y_t), y_t) + \text{dist}_G(y_t, v) + \text{dist}_G(v, q(v)) \leq 2\,\text{dist}_G(y_t, v)$. □

By Claim 6.2.6, Claim 6.2.7, and Claim 6.2.8, we have:

$$\begin{aligned}
\text{dist}_H(u, v) &\leq w'(q(y_t), v) + \sum_{i=1}^{t} (w'(q(y_{i-1}), q(x_i)) + w'(q(x_i), q(y_i))) \\
&\leq 2\,\text{dist}_G(y_t, v) + \sum_{i=1}^{t} (4r_{G,b}(y_{i-1}) + 4\,\text{dist}_G(y_{i-1}, x_i) + 2w(x_i, y_i)) \\
&\leq 4\,\text{dist}_G(u, v) + 4\sum_{i=1}^{t} r_{G,b}(y_{i-1}) \\
&\leq 8\,\text{dist}_G(u, v), \qquad (6.5)
\end{aligned}$$

where the third inequality follows from Equation (6.4), and the last inequality follows from Claim 6.2.4.

Next, we will show that H is actually a strong subemulator of G, and $q : V \to V'$ is a corresponding leader mapping. By line 3, $\forall v \in V$, we have $q(v) \in \mathrm{Ball}_{G,b}(v) \cap V'$. Consider two arbitrary vertices $u, v \in V$. There are two cases. In the first case, $u \in \mathrm{Ball}_{G,b}(v)$ or $v \in \mathrm{Ball}_{G,b}(u)$. In this case, we have $\mathrm{dist}_H(q(u), q(v)) \le w'(q(u), q(v)) \le \mathrm{dist}_G(q(u), u) + \mathrm{dist}_G(u, v) + \mathrm{dist}_G(v, q(v))$, where the last inequality follows from line 9. In the second case, neither $u \in \mathrm{Ball}_{G,b}(v)$ nor $v \in \mathrm{Ball}_{G,b}(u)$. In this case, since $q(u) \in \mathrm{Ball}_{G,b}(v), q(v) \in \mathrm{Ball}_{G,b}(u)$, we know that $\mathrm{dist}_G(u, v) \ge \max(\mathrm{dist}_G(v, q(v)), \mathrm{dist}_G(u, q(u)))$. Thus, we have

$$\begin{aligned}
\mathrm{dist}_H(q(u), q(v)) &\le 8\,\mathrm{dist}_G(q(u), q(v)) \\
&\le 8(\mathrm{dist}_G(q(u), u) + \mathrm{dist}_G(u, v) + \mathrm{dist}(v, q(v))) \\
&= \mathrm{dist}_G(q(u), u) + \mathrm{dist}_G(q(v), v) + (7\,\mathrm{dist}_G(q(u), u) + 7\,\mathrm{dist}_G(q(v), v) + 8\,\mathrm{dist}_G(u, v)) \\
&\le \mathrm{dist}_G(q(u), u) + \mathrm{dist}_G(q(v), v) + 22\,\mathrm{dist}_G(u, v),
\end{aligned}$$

where the first inequality follows from Equation (6.5), the second inequality follows from triangle inequality, and the last inequality follows from $\mathrm{dist}_G(u, v) \ge \max(\mathrm{dist}_G(v, q(v)), \mathrm{dist}_G(u, q(u)))$. □

Algorithm 33 Construction of the Subemulator

```
1: procedure SUBEMULATOR(G = (V, E, w), b ∈ [|V|])
2:     Output: H = (V', E', w'), q : V → V'
3:     V' ←SAMPLES(G, b).                          ▷ Algorithm 31.
4:     H, q ←CONNECTS(G, V', b).                   ▷ Algorithm 32.
5:     Return H, q.
6: end procedure
```

Theorem 6.2.9 (Construction of the subemulator)**.** *Consider a connected n-vertex m-edge undirected weighted graph $G = (V, E, w)$ and a parameter $b \in [n]$.* SUBEMULATOR(G, b) *(Algorithm 33) will output an undirected weighted graph $H = (V', E', w')$ and $q : V \to V'$ such that H is a strong $(8, b, 22)$-subemulator of G, and q is a corresponding leader mapping (Definition 6.2.1). Furthermore, $\mathbf{E}[|V'|] \le \min(75\log(n)/b, 3/4)n$, $|E'| \le m + nb$.*

Proof. Follows directly from Lemma 6.2.2 and Lemma 6.2.3. □

6.2.2 A warm-up algorithm: distance oracle via subemulator

Given a weighted undirected graph, a distance oracle is a static data structure which uses small space and can be used to efficiently return an approximate distance between any pair of query vertices. In this section, we give a warm-up algorithm which is a direct application of subemulator. In section 6.2.3, we will show how to apply the preprocessing procedure PREPROC (Algorithm 34) in our construction of low hop emulator.

Algorithm 34 Distance Oracle

procedure PREPROC($G = (V, E, w), k$)
  $n \leftarrow |V|, m \leftarrow |E|$.
  $t \leftarrow 0, H_0 = (V_0, E_0, w_0) \leftarrow G, b_0 \leftarrow \max\left(\lceil (75 \log n)^2 \rceil, n^{1/(2k)}\right)$.
  $n_0 \leftarrow |V_0|, m_0 \leftarrow |E_0|$
  while $n_t \geq b_t$ **do**
    $H_{t+1} = (V_{t+1}, E_{t+1}, w_{t+1}), q_t \leftarrow$SUBEMULATOR$(H_t, b_t)$. ▷ See Algorithm 33.
    $\forall v \in V_t$, let $B_t(v) \leftarrow \text{Ball}_{H_t, b_t}(v)$ and compute and store $\text{dist}_{H_t}(v, u)$ for every $u \in B_t(v)$.
    $n_{t+1} \leftarrow |V_{t+1}|, m_{t+1} \leftarrow |E_{t+1}|$.
    $b_{t+1} \leftarrow b_t^{1.25}$.
    $t \leftarrow t + 1$.
  end while
  For $v \in V_t$, $B_t(v) \leftarrow V_t$, compute $\text{dist}_{H_t}(v, u)$ for $u \in V_t$, and $q_t(v) \leftarrow x$ where $x \in V_t$ is smallest.
end procedure
procedure QUERY(u, v)
  Output: $d \in \mathbb{Z}_{\geq 0}$
  $l \leftarrow 0, d_0 \leftarrow 0, u_0 \leftarrow u, v_0 \leftarrow v$.
  while $v_l \notin B_l(u_l)$ and $u_l \notin B_l(v_l)$ **do**
    $d_l \leftarrow \text{dist}_{H_l}(u_l, q_l(u_l)) + \text{dist}_{H_l}(v_l, q_l(v_l))$.
    $u_{l+1} = q_l(u_l), v_{l+1} = q_l(v_l)$.
    $l \leftarrow l + 1$
  end while
  $d_l \leftarrow \text{dist}_{H_l}(u_l, v_l)$.
  Return $d = \sum_{i=0}^{l} d_i$.
end procedure

Lemma 6.2.10 (Properties of the preprocessing algorithm)**.** *Given a connected weighted graph $G = (V, E, w)$ with $|V| = n$, $|E| = m$, and a parameter $k \in [0.5, 0.5 \log n]$, let t be the value at the end of* PREPROC(G, k) *(Algorithm 34). For $i > t$, define $n_i = m_i = 0, b_i = b_{i-1}^{1.25}, V_i = \emptyset$. We have following properties:*

1. $t \le 4\lceil \log(k) + 1 \rceil$.

2. *For* $i \in \mathbb{Z}_{\ge 0}$,

   - $\mathbf{E}[n_i] \le \max(n^{1+1/k}, n \cdot (75 \log n)^4)/b_i^2$,
   - $\mathbf{E}[m_i] \le m + 2 \cdot \max(n^{1+1/(2k)}, n \cdot (75 \log n)^2)$,
   - $\mathbf{E}\left[\sum_{v \in V_i} |B_i(v)|\right] \le \max(n^{1+1/k}, n \cdot (75 \log n)^4)/b_i$.

Proof. By line 9 and the definition of b_i for $i > t$, we have $\forall i \in \mathbb{Z}_{\ge 0}, b_i = b_0^{1.25^i}$.

Consider $i = 1 + \lceil \log_{1.25} \log_{b_0} n \rceil$. We have $b_i = b_0^{1.25^i} > n$. By line 5, we can conclude $t < i$. Thus, $t \le \lceil \log_{1.25} \log_{b_0} n \rceil \le \lceil (\log 2k)/\log 1.25 \rceil \le 4\lceil \log(k) + 1 \rceil$.

Consider n_i, we have $\forall i \in \mathbb{Z}_{\ge 1}$,

$$\begin{aligned}
\mathbf{E}[n_i] &\le (75 \log n)/b_{i-1} \cdot \mathbf{E}[n_{i-1}] \\
&\le \mathbf{E}[n_{i-1}]/b_{i-1}^{0.5} \\
&\le n/\left(\prod_{j=0}^{i-1} b_j\right)^{0.5} \\
&\le n/\left(b_0^{\sum_{j=0}^{i-1} 1.25^j}\right)^{0.5} \\
&= n/b_0^{(1.25^i - 1)\cdot 2} \\
&= n/b_i^2 \cdot b_0^2 \\
&\le \max(n^{1+1/k}, n \cdot (75 \log n)^4)/b_i^2, \qquad (6.6)
\end{aligned}$$

where the first inequality follows from Theorem 6.2.9, the second inequality follows from that b_i is increasing and $b_0^{0.5} \ge 75 \log n$, the forth inequality follows from $\forall j \in \mathbb{Z}_{\ge 0}, b_j = b_0^{1.25^j}$.

Consider m_i, we have $\forall i \in \mathbb{Z}_{\ge 1}$,

$$\mathbf{E}[m_i] \le \mathbf{E}[m_{i-1}] + \mathbf{E}[n_{i-1}] \cdot b_i$$

$$
\begin{aligned}
&= m + \sum_{j=0}^{i-1} \mathbf{E}[n_j] \cdot b_j \\
&\leq m + \sum_{j=0}^{i-1} \max(n^{1+1/k}, n \cdot (75 \log n)^4)/b_j \\
&\leq m + 2\max(n^{1+1/(2k)}, n \cdot (75 \log n)^2),
\end{aligned}
$$

where the first inequality follows from Theorem 6.2.9, the second inequality follows from Equation (6.6), and the last inequality follows from $b_{j+1} \geq 2b_j$ and $b_0 = \max(n^{1/(2k)}, (75 \log n)^2)$.

Consider $\sum_{v \in V_i} |B_i(v)|$, we have $\forall i \in \mathbb{Z}_{\geq 1}$,

$$
\mathbf{E}\left[\sum_{v \in V_i} |B_i(v)|\right] \leq \mathbf{E}\left[\sum_{v \in V_i} b_i\right] = E[n_i] \cdot b_i \leq \max(n^{1+1/k}, n \cdot (75 \log n)^4)/b_i,
$$

where the first inequality follows from line 7, line 12 and the definition of Ball_{H_i, b_i}, and the last inequality follows from Equation (6.6). □

Lemma 6.2.11 (Correctness of the query algorithm)**.** *Given a connected weighted graph $G = (V, E, w)$ with $|V| = n$, $|E| = m$, and a parameter $k \in [0.5, 0.5 \log n]$, run preprocessing* PREPROC(G, k) *(Algorithm 34). Then $\forall u, v \in V$, the output d of* QUERY(u, v) *(Algorithm 34) satisfies* $\text{dist}_G(u, v) \leq d \leq 26^{4\lceil \log(k)+1 \rceil} \text{dist}_G(u, v)$. *The running time of* QUERY(u, v) *is* $O(\log(4k))$.

Proof. Let t be the value at the end of the preprocessing procedure PREPROC(G, k). Let l be the value at the end of the query procedure QUERY(u, v). By induction, we can show that $\forall i \in \{0, 1, \cdots, l\}, u_i, v_i \in V_i$. Since $\forall v \in V_t, B_t(v) = V_t$ and the condition of line 17, we have $l \leq t$, i.e., the query procedure will terminate. By Lemma 6.2.10, we have $t \leq 4\lceil \log(k) + 1 \rceil$. Thus, the running time of QUERY(u, v) is $O(\log(4k))$.

In the following we will show that $\forall i \in \{0, 1, \cdots, l\}, \text{dist}_{H_i}(u_i, v_i) \leq \sum_{j=i}^{l} d_j \leq 26^{l-i} \text{dist}_{H_i}(u_i, v_i)$. Our proof is by induction. The base case is $i = l$. By line 22, $d_l = \text{dist}_{H_l}(u_l, v_l)$. Suppose

$\text{dist}_{H_{i+1}}(u_{i+1}, v_{i+1}) \leq \sum_{j=i+1}^{l} d_j \leq 26^{l-i-1} \text{dist}_{H_{i+1}}(u_{i+1}, v_{i+1})$. For the contraction,

$$\begin{aligned}
\sum_{j=i}^{l} d_j &= \text{dist}_{H_i}(u_i, u_{i+1}) + \text{dist}_{H_i}(v_i, v_{i+1}) + \sum_{j=i+1}^{l} d_j \\
&\geq \text{dist}_{H_i}(u_i, u_{i+1}) + \text{dist}_{H_i}(v_i, v_{i+1}) + \text{dist}_{H_{i+1}}(u_{i+1}, v_{i+1}) \\
&\geq \text{dist}_{H_i}(u_i, u_{i+1}) + \text{dist}_{H_i}(v_i, v_{i+1}) + \text{dist}_{H_i}(u_{i+1}, v_{i+1}) \\
&\geq \text{dist}_{H_i}(u_i, v_i),
\end{aligned}$$

where the first equality follows from line 18, the first inequality follows from $\sum_{j=i+1}^{l} d_j \geq \text{dist}_{H_{i+1}}(u_{i+1}, v_{i+1})$, the second inequality follows from Theorem 6.2.9 that H_{i+1} is a subemulator of H_i and Definition 6.2.1, and the last inequality follows from triangle inequality.

For the expansion,

$$\begin{aligned}
\sum_{j=i}^{l} d_j &= \text{dist}_{H_i}(u_i, u_{i+1}) + \text{dist}_{H_i}(v_i, v_{i+1}) + \sum_{j=i+1}^{l} d_j \\
&\leq \text{dist}_{H_i}(u_i, u_{i+1}) + \text{dist}_{H_i}(v_i, v_{i+1}) + 26^{l-i-1} \text{dist}_{H_{i+1}}(u_{i+1}, v_{i+1}) \\
&\leq \text{dist}_{H_i}(u_i, u_{i+1}) + \text{dist}_{H_i}(v_i, v_{i+1}) + 8 \cdot 26^{l-i-1} \text{dist}_{H_i}(u_{i+1}, v_{i+1}) \\
&\leq \text{dist}_{H_i}(u_i, u_{i+1}) + \text{dist}_{H_i}(v_i, v_{i+1}) + 8 \cdot 26^{l-i-1}(\text{dist}_{H_i}(u_{i+1}, u_i) + \text{dist}_{H_i}(u_i, v_i) + \text{dist}_{H_i}(v_i, v_{i+1})) \\
&= (8 \cdot 26^{l-i-1} + 1)(\text{dist}_{H_i}(u_i, u_{i+1}) + \text{dist}_{H_i}(v_i, v_{i+1})) + 8 \cdot 26^{l-i-1} \text{dist}_{H_i}(u_i, v_i) \\
&\leq 24 \cdot 26^{l-i-1} \text{dist}_{H_i}(u_i, v_i) + 2 \, \text{dist}_{H_i}(u_i, v_i) \\
&\leq 26 \cdot 26^{l-i-1} \text{dist}_{H_i}(u_i, v_i) = 26^{l-i} \text{dist}_{H_i}(u_i, v_i),
\end{aligned}$$

where the first equality follows from line 18, the first inequality follows from

$$\sum_{j=i+1}^{l} d_j \leq 26^{l-i-1} \text{dist}_{H_{i+1}}(u_{i+1}, v_{i+1}),$$

the second inequality from Theorem 6.2.9 that H_{i+1} is an $(8, b_i)$-subemulator of H_i and Definition 6.2.1, the third inequality follows from triangle inequality, and the forth inequality follows

from that $\text{dist}_{H_i}(u_i, v_i) \geq \max(\text{dist}_{H_i}(u_i, u_{i+1}), \text{dist}_{H_i}(v_i, v_{i+1}))$ since neither $u_i \in \text{Ball}_{H_i,b_i}(v_i)$ nor $v_i \in \text{Ball}_{H_i,b_i}(u_i)$.

By Lemma 6.2.10, we have $t \leq 4\lceil \log(k) + 1 \rceil$. Since $l \leq t \leq 4\lceil \log(k) + 1 \rceil$ and $\text{dist}_G(u, v) = \text{dist}_{H_0}(u_0, v_0)$, $\text{dist}_G(u, v) \leq d \leq 26^{4\lceil \log(k)+1 \rceil} \text{dist}_G(u, v)$. □

6.2.3 Low hop emulator

Distance preserving graph with low hop diameter

In this section, we construct a new graph with more vertices and edges such that the distance is approximately preserved and there always exists a low hop shortest path between any pair of vertices in the new graph. In the next section, we will show how to refine the construction to make the new graph be an emulator.

Lemma 6.2.12 (Size of the graph)**.** *Consider a connected undirected weighted graph $G = (V, E, w)$ and $k \in [0.5, 0.5 \log n]$, where $n = |V|$. Let $G' = (V', E', w')$ be the output of* LowHopDimGraph(G, k) *(Algorithm 35). Then,* $\mathbf{E}[|V'|] \leq O(n)$, $\mathbf{E}[|E'|] \leq O((n^{1+1/(2k)} + n \log^2 n + m) \log k)$.

Proof. For $i \in \{0, 1, \cdots, t\}$, let $n_i = |V_i|, m_i = |E_i|$. We have

$$\mathbf{E}[|V'|] = \sum_{i=0}^{t} \mathbf{E}[n_i] \leq \sum_{i=0}^{t} \max(n^{1+1/k}, n(75 \log n)^4)/b_i^2 \leq \sum_{i=0}^{t} \max(n^{1+1/k}, n(75 \log n)^4)/(b_0^2 \cdot 2^i) \leq 2n, \tag{6.7}$$

where the first inequality follows from Lemma 6.2.10, the second inequality follows from $b_i/b_0 > 2^i$, and the last inequality follows from $b_0 = \max\left((75 \log n)^2, n^{1/(2k)}\right)$.

Consider $|E'|$, we have

$$\begin{aligned}\mathbf{E}[|E'|] &\leq \sum_{i=0}^{t-1} \mathbf{E}[n_i] + \sum_{i=0}^{t} \mathbf{E}\left[\sum_{v \in V_i} |B_i(v)|\right] + \sum_{i=0}^{t} \mathbf{E}[m_i] \\ &\leq 2n + \sum_{i=0}^{t} \mathbf{E}\left[\sum_{v \in V_i} |B_i(v)|\right] + \sum_{i=0}^{t} \mathbf{E}[m_i]\end{aligned}$$

Algorithm 35 Low Hop Diameter Distance Preserving Graph

1: **procedure** LOWHOPDIMGRAPH($G = (V, E, w), k$)
2: Output: $G' = (V', E', w')$
3: Run the processing procedure PREPROC(G, k), and let t be the value at the end of the procedure. $\forall i \in \{0, 1, \cdots, t\}$, let $H_i = (V_i, E_i, w_i)$, $q_i : V_i \to V_{i+1}, B_i : V_i \to 2^{V_i}$, b_i be computed by the such procedure. ▷ See Algorithm 34.
4: Initialize $V' \leftarrow \emptyset, E' \leftarrow \emptyset$.
5: For $v \in V$, if $v \in V_i, v \notin V_{i+1}$, add $i + 1$ copies of v into V', i.e., $V' \leftarrow V' \cup \{v^{(0)}, v^{(1)}, \cdots, v^{(i)}\}$.
6: For $i \in \{0, 1, \cdots, t-1\}$, for each $v \in V_i$, $E' \leftarrow E' \cup \{\{v^{(i)}, u^{(i+1)}\}\}$, where $u = q_i(v)$.
7: For $i \in \{0, 1, \cdots, t\}$, for each $\{u, v\} \in E_i$, $E' \leftarrow E' \cup \{\{u^{(i)}, v^{(i)}\}\}$.
8: For $i \in \{0, 1, \cdots, t\}$, for each $v \in V_i$, for each $u \in B_i(v)$, $E' \leftarrow E' \cup \{\{u^{(i)}, v^{(i)}\}\}$.
9: For each $e' \in E'$, initialize $w'(e') \leftarrow \infty$.
10: For $i \in \{0, 1, \cdots, t-1\}$, for each $v \in V_i$, consider $e' = \{v^{(i)}, u^{(i+1)}\}$ where $u = q_i(v)$. Let

$$w'(e') \leftarrow \min(w'(e'), 27^{t-i-1} \cdot \mathrm{dist}_{H_i}(u, v)).$$

11: For $i \in \{0, 1, \cdots, t\}$, for each $\{u, v\} \in E_i$, consider $e' = \{u^{(i)}, v^{(i)}\}$. Let

$$w'(e') \leftarrow \min(w'(e'), 27^{t-i} \cdot w_i(u, v)).$$

12: For $i \in \{0, 1, \cdots, t\}$, for each $v \in V_i$, for each $u \in B_i(v)$, consider $e' = \{u^{(i)}, v^{(i)}\}$. Let

$$w'(e') \leftarrow \min(w'(e'), 27^{t-i} \cdot \mathrm{dist}_{H_i}(u, v)).$$

13: Output $G' = (V', E', w')$.
14: **end procedure**

$$\begin{aligned}
&\leq 2n + \sum_{i=0}^{t} \max(n^{1+1/k}, n \cdot (75 \log n)^4)/b_i + \sum_{i=0}^{t} \mathbf{E}[m_i] \\
&\leq 2n + \sum_{i=0}^{t} \max(n^{1+1/k}, n \cdot (75 \log n)^4)/(2^i \cdot b_0) + \sum_{i=0}^{t} \mathbf{E}[m_i] \\
&\leq 2n + 2 \cdot \max(n^{1+1/(2k)}, n \cdot (75 \log n)^2) + \sum_{i=0}^{t} \mathbf{E}[m_i] \\
&\leq 2n + 2 \cdot \max(n^{1+1/(2k)}, n \cdot (75 \log n)^2) + 4(\lceil \log(k) + 1 \rceil \cdot (m + 2 \cdot \max(n^{1+1/(2k)}, n \cdot (75 \log n)^2)) \\
&\leq 8\lceil \log(k) + 1 \rceil \cdot (m + 2 \cdot \max(n^{1+1/(2k)}, n \cdot (75 \log n)^2)),
\end{aligned}$$

where the first inequality follows from line 6, line 7 and line 8, the second inequality follows from Equation (6.7), the third inequality follows from Lemma 6.2.10, the forth inequality follows from

$b_i > b_0 \cdot 2^i$, the sixth inequality follows from that $t \leq 4\lceil \log(k) + 1 \rceil$ and

$$\mathbf{E}[m_i] \leq m + 2 \cdot \max(n^{1+1/(2k)}, n \cdot (75 \log n)^2)$$

by Lemma 6.2.10. □

Lemma 6.2.13 (Low hop diameter)**.** *Consider a connected undirected weighted graph $G = (V, E, w)$ and $k \in [0.5, 0.5 \log n]$, where $n = |V|$. Let $G' = (V', E', w')$ be the output of* LowHopDimGraph(G, k) *(Algorithm 35). Then, $\forall u, v \in V$,*

$$\mathrm{dist}_G(u, v) \leq \mathrm{dist}_{G'}(u^{(0)}, v^{(0)}) \leq 27^{4\lceil \log(k)+1 \rceil} \cdot \mathrm{dist}_G(u, v).$$

Furthermore, $\forall x, y \in V'$,

$$\mathrm{dist}_{G'}(x, y) = \mathrm{dist}_{G'}^{(16\lceil \log(k)+1 \rceil)}(x, y).$$

Proof. Consider two vertices $u, v \in V$. Since $H_0 = G$ and the construction of the weights in line 11, we have $\mathrm{dist}_{G'}(u^{(0)}, v^{(0)}) \leq 27^t \cdot \mathrm{dist}_G(u, v)$. Due to Lemma 6.2.10, $t \leq 4\lceil \log(k) + 1 \rceil$. We have $\mathrm{dist}_{G'}(u^{(0)}, v^{(0)}) \leq 27^{4\lceil \log(k)+1 \rceil} \cdot \mathrm{dist}_G(u, v)$. Because $\forall i \in [t]$, H_i is a subemulator of H_{i-1}, we have $\mathrm{dist}_{G'}(u^{(0)}, v^{(0)}) \geq \mathrm{dist}_G(u, v)$ by Definition 6.2.1 and the construction of the weights in line 10, line 11 and line 12.

For a vertex $v^{(i)} \in V'$, we define the level of $v^{(i)}$ as i, i.e., $\mathrm{level}(v^{(i)}) = i$. Consider two arbitrary vertex $x, y \in V'$. Let $p = (x = z_0, z_1, z_2, \cdots, z_h = y)$ be a shortest path with minimum number of hops between x and y in G'.

Claim 6.2.14. $\forall j \in [h]$, $|\mathrm{level}(z_{j-1}) - \mathrm{level}(z_j)| \leq 1$.

Proof. It follows directly from the construction of E' in Algorithm 35. □

Claim 6.2.15. $\forall j \in [h-1]$, *either* $\mathrm{level}(z_{j-1}) \neq \mathrm{level}(z_j)$ *or* $\mathrm{level}(z_j) \neq \mathrm{level}(z_{j+1})$.

Proof. We prove by contradiction. Suppose $\text{level}(z_{j-1}) = \text{level}(z_j) = \text{level}(z_{j+1}) = c$. Suppose z_{j-1}, z_j, z_{j+1} are copies of u, a, v respectively, i.e, $z_{j-1} = u^{(c)}, z_j = a^{(c)}, z_{j+1} = v^{(c)}$. By the construction, we know that $w'(z_{j-1}, z_j) \geq 27^{t-c} \operatorname{dist}_{H_c}(u, a)$ and $w'(z_j, z_{j+1}) \geq 27^{t-c} \operatorname{dist}_{H_c}(a, v)$. Thus, $w'(z_{j-1}, z_j) + w'(z_j, z_{j+1}) \geq 27^{t-c} \operatorname{dist}_{H_c}(u, v)$. There are two cases. In the first case, either $v \in B_c(u)$ or $u \in B_c(v)$. In this case, $\{z_{j-1}, z_{j+1}\} \in E'$ by line 8 and $w'(z_{j-1}, z_{j+1}) \leq 27^{t-c} \operatorname{dist}_{H_c}(u, v)$ by line 12. Thus, $\{z_{j-1}, z_{j+1}\} \in E', w'(z_{j-1}, z_{j+1}) \leq w'(z_{j-1}, z_j) + w'(z_j, z_{j+1})$, and it contradicts to that p has the minimum number of hops. In the second case, neither $v \in \text{Ball}_{H_c, b_c}(u)$ nor $u \in \text{Ball}_{H_c, b_c}(v)$. Consider this case. Let $u' = q_c(u), v' = q_c(v)$. We have

$$
\begin{aligned}
& w'(z_{j-1}, u'^{(c+1)}) + \operatorname{dist}_{G'}(u'^{(c+1)}, v'^{(c+1)}) + w'(v'^{(c+1)}, z_{j+1}) \\
\leq\ & 27^{t-c-1} \cdot (\operatorname{dist}_{H_c}(u, u') + \operatorname{dist}_{H_{c+1}}(u', v') + \operatorname{dist}_{H_c}(v', v)) \\
\leq\ & 27^{t-c-1} \cdot (2 \operatorname{dist}_{H_c}(u, v) + \operatorname{dist}_{H_{c+1}}(u', v')) \\
\leq\ & 27^{t-c-1} \cdot (2 \operatorname{dist}_{H_c}(u, v) + 8 \operatorname{dist}_{H_c}(u', v')) \\
\leq\ & 27^{t-c-1} \cdot (2 \operatorname{dist}_{H_c}(u, v) + 8(\operatorname{dist}_{H_c}(u, u') + \operatorname{dist}_{H_c}(u, v) + \operatorname{dist}_{H_c}(v', v))) \\
\leq\ & 27^{t-c-1} \cdot 26 \operatorname{dist}_{H_c}(u, v) \\
<\ & 27^{t-c} \operatorname{dist}_{H_c}(u, v) \\
\leq\ & w'(z_{j-1}, z_j) + w'(z_j, z_{j+1}),
\end{aligned}
$$

where the first inequality follows from the construction of the edges and the corresponding weights in G', the second inequality follows from that $\operatorname{dist}_{H_c}(u, v) \geq \max(\operatorname{dist}_{H_c}(u, u'), \operatorname{dist}_{H_c}(v, v'))$ implied by neither $v \in \text{Ball}_{H_c, b_c}(u)$ nor $u \in \text{Ball}_{H_c, b_c}(v)$, the third inequality follows from H_{c+1} is a $(8, b_c)$-subemulator of H_c (see Theorem 6.2.9 and Definition 6.2.1), the forth inequality follows from triangle inequality, the fifth inequality follows from $\operatorname{dist}_{H_c}(u, v) \geq \max(\operatorname{dist}_{H_c}(u, u'), \operatorname{dist}_{H_c}(v, v'))$ again. In this case, we can find a shorter path which contradicts to that p is the shortest path. □

Claim 6.2.16. $\forall j \in \{0, 1, \cdots, h-2\}$, *if* $\text{level}(z_{j+1}) < \text{level}(z_j)$, *then* $\forall j' \in \{j+2, \cdots, h\}, \text{level}(z_{j'}) < \text{level}(z_j)$.

Proof. We prove it by contradiction. Suppose $\text{level}(z_{j+1}) < \text{level}(z_j)$ and $\exists j'' > j$ such that $\text{level}(z_{j''}) \geq \text{level}(z_j)$. We can find an $z_{j'}$ such that $\text{level}(z_{j'})$ is the minimum among $\text{level}(z_j), \text{level}(z_{j+1}), \cdots, \text{level}(z_{j''})$. Let $f < j'$ be the largest value such that $\text{level}(z_f) > \text{level}(z_{j'})$. Let $g > j'$ be the smallest value such that $\text{level}(z_g) > \text{level}(z_{j'})$. Due to Claim 6.2.14, we have $\text{level}(z_f) = \text{level}(z_{f+1}) + 1 = \cdots = \text{level}(z_{j'}) + 1 = \cdots = \text{level}(z_{g-1}) + 1 = \text{level}(z_g)$. Suppose $\text{level}(z_{j'}) = c$. Let z_{f+1}, z_{g-1} be the copies of u, v respectively, i.e., $z_{f+1} = u^{(c)}$ and $z_{g-1} = v^{(c)}$. Let $u' = q_c(u), v' = q_c(v)$. Then $z_f = u'^{(c+1)}$ and $z_g = v'^{(c+1)}$. We have

$$\begin{aligned}
\text{dist}_{G'}(u', v') &= \sum_{a=f}^{g-1} w'(z_a, z_{a+1}) \\
&= 27^{t-c-1} \text{dist}_{H_c}(u', u) + 27^{t-c} \text{dist}_{H_c}(u, v) + 27^{t-c-1} \text{dist}_{H_c}(v, v') \\
&= 27^{t-c-1}(\text{dist}_{H_c}(u', u) + \text{dist}_{H_c}(v', v) + 27 \text{dist}_{H_c}(u, v)).
\end{aligned}$$

According to Theorem 6.2.9, H_{c+1} is a strong $(8, b_c, 22)$-subemulator. By Definition 6.2.1, we have $\text{dist}_{H_{c+1}}(u', v') \leq \text{dist}_{H_c}(u', u) + \text{dist}_{H_c}(v', v) + 22 \text{dist}_{H_c}(u, v)$. Thus, we have

$$\text{dist}_{G'}(u', v') \leq 27^{t-c-1} \text{dist}_{H_{c+1}}(u', v') \leq 27^{t-c-1}(\text{dist}_{H_c}(u', u) + \text{dist}_{H_c}(v', v) + 22 \text{dist}_{H_c}(u, v))$$

which leads to a contradiction. □

By Claim 6.2.16, there must exist $j \in [h]$ such that $\forall j' \in \{0, 1, \cdots, j-1\}, \text{level}(z_{j'}) \leq \text{level}(z_{j'+1})$ and $\forall j' \in \{j, j+1, \cdots, h-1\}, \text{level}(z_{j'}) \geq \text{level}(z_{j'+1})$. Together with Claim 6.2.15, we can conclude that $h \leq 4 \cdot t \leq 16\lceil \log(k) + 1 \rceil$ where the last inequality follows from $t \leq 4\lceil \log(k) + 1 \rceil$ by lemma 6.2.10. □

Low hop emulator

In this section, we show how to simplify Algorithm 35 to obtain an emulator which has smaller size than the graph obtained in the previous section. We have two observations. The first observation is that line 7 and line 11 of Algorithm 35 are useless.

Observation 6.2.17. *Consider a connected undirected weighted graph $G = (V, E, w)$ and $k \in [0.5, 0.5 \log n]$, where $n = |V|$. Let $G' = (V', E', w')$ be the output of* LowHopDimGraph(G, k) *(Algorithm 35), and let $t, H_i = (V_i, E_i, w_i), B_i(\cdot)$ be the same as described in Algorithm 35. Then for any $i \in \{0, 1, \cdots, t\}$ and for any $\{u, v\} \in E_i$, we have either $\mathrm{dist}_{G'}(u^{(i)}, v^{(i)}) < 27^{t-i} \cdot w_i(u, v)$, $u \in B_i(v)$, or $v \in B_i(u)$.*

Proof. Suppose $u \notin B_i(v)$ and $v \notin B_i(u)$. Let $u' = q_i(u), v' = q_i(v)$. We have

$$\begin{aligned}
\mathrm{dist}_{G'}(u^{(i)}, v^{(i)}) &\le w'(u^{(i)}, u'^{(i+1)}) + \mathrm{dist}_{G'}(u'^{(i+1)}, v'^{(i+1)}) + w'(v'^{(i+1)}, v^{(i)}) \\
&\le 27^{t-i-1} \cdot (\mathrm{dist}_{H_i}(u, u') + \mathrm{dist}_{H_{i+1}}(u', v') + \mathrm{dist}_{H_i}(v', v)) \\
&\le 27^{t-i-1} \cdot (2\,\mathrm{dist}_{H_i}(u, v) + \mathrm{dist}_{H_{i+1}}(u', v')) \\
&\le 27^{t-i-1} \cdot (2\,\mathrm{dist}_{H_i}(u, v) + 8\,\mathrm{dist}_{H_i}(u', v')) \\
&\le 27^{t-i-1} \cdot (2\,\mathrm{dist}_{H_i}(u, v) + 8(\mathrm{dist}_{H_i}(u, u') + \mathrm{dist}_{H_i}(u, v) + \mathrm{dist}_{H_i}(v', v))) \\
&\le 27^{t-i-1} \cdot 26\,\mathrm{dist}_{H_i}(u, v) \\
&< 27^{t-i} \cdot \mathrm{dist}_{H_i}(u, v) \\
&\le 27^{t-i} \cdot w_i(u, v),
\end{aligned}$$

where the second inequality follows from the construction of the edges and their corresponding weights in Algorithm 35, the third inequality follows from $\mathrm{dist}_{H_i}(u, u'), \mathrm{dist}_{H_i}(v', v) \le \mathrm{dist}_{H_i}(u, v)$ since $v \notin B_i(u)$ and $u \notin B_i(v)$, the forth inequality follows from that H_{i+1} is a $(8, b_i)$-subemulator of H_i (see Theorem 6.2.9 and Definition 6.2.1), the fifth inequality follows from triangle inequality, the sixth inequality follows from $\mathrm{dist}_{H_i}(u, u'), \mathrm{dist}_{H_i}(v', v) \le \mathrm{dist}_{H_i}(u, v)$ again. □

If $u \in B_i(v)$ or $v \in B_i(u)$, then by line 8 and line 12 of Algorithm 35, there is an edge in G' with weight at most $27^{t-i} \cdot \mathrm{dist}_{H_i}(u, v) \le 27^{t-i} \cdot w_i(u, v)$. Otherwise $\mathrm{dist}_{G'}(u^{(i)}, v^{(i)}) < 27^{t-i} \cdot w_i(u, v)$. Together with the above observation, we can avoid using the edges constructed by line 7 of Algorithm 35 in any shortest path. The second observation is that $\forall i \in [t], v \in V_i$, $w'(v^{(i-1)}, v^{(i)}) = 0$. Thus, we can use a single vertex v to denote $v^{(0)}, v^{(1)}, \cdots, v^{(i)}$.

By removing line 7 and line 11 of Algorithm 35, and contraction of the vertices from different levels, we obtain Algorithm 36.

Algorithm 36 Low Hop Emulator

1: **procedure** LOWHOPDIMEMULATOR($G = (V, E, w), k$)
2: Output: $G' = (V', E', w')$
3: Run the processing procedure PREPROC(G, k), and let t be the value at the end of the procedure. $\forall i \in \{0, 1, \cdots, t\}$, let $H_i = (V_i, E_i, w_i)$, $q_i : V_i \to V_{i+1}, B_i : V_i \to 2^{V_i}$, b_i be computed by the such procedure. ▷ See Algorithm 34.
4: Initialize $E' \leftarrow \emptyset$.
5: For $i \in \{0, 1, \cdots, t-1\}$, for each $v \in V_i$, $E' \leftarrow E' \cup \{\{v, u\}\}$, where $u = q_i(v)$.
6: For $i \in \{0, 1, \cdots, t\}$, for each $v \in V_i$, for each $u \in B_i(v)$, $E' \leftarrow E' \cup \{\{u, v\}\}$.
7: For each $e' \in E'$, initialize $w'(e') \leftarrow \infty$.
8: For $i \in \{0, 1, \cdots, t-1\}$, for each $v \in V_i$, consider $e' = \{v, u\}$ where $u = q_i(v)$. Let

$$w'(e') \leftarrow \min(w'(e'), 27^{t-i-1} \cdot \mathrm{dist}_{H_i}(u, v)).$$

9: For $i \in \{0, 1, \cdots, t\}$, for each $v \in V_i$, for each $u \in B_i(v)$, consider $e' = \{u, v\}$. Let

$$w'(e') \leftarrow \min(w'(e'), 27^{t-i} \cdot \mathrm{dist}_{H_i}(u, v)).$$

10: Output $G' = (V, E', w')$.
11: **end procedure**

Theorem 6.2.18 (Low hop emulator)**.** *Consider a connected undirected weighted graph $G = (V, E, w)$ and $k \in [0.5, 0.5 \log n]$. Let $G' = (V, E', w')$ be the output of* LOWHOPDIMEMULATOR(G, k) *(Algorithm 36). Then, $\mathbf{E}[|E'|] \leq O(n^{1+1/(2k)} + n \log^2 n)$, where $n = |V|$. Furthermore, $\forall u, v \in V$,*

$$\mathrm{dist}_G(u, v) \leq \mathrm{dist}_{G'}(u, v) \leq 27^{4\lceil \log(k)+1 \rceil} \cdot \mathrm{dist}_G(u, v).$$

Furthermore, $\forall u, v \in V$,

$$\mathrm{dist}_{G'}(u, v) = \mathrm{dist}_{G'}^{(16\lceil \log(k)+1 \rceil)}(u, v).$$

Proof. Consider $|E'|$, we have

$$\begin{aligned}
\mathbf{E}[|E'|] &\leq \sum_{i=0}^{t-1} \mathbf{E}[n_i] + \sum_{i=0}^{t} \mathbf{E}\left[\sum_{v \in V_i} |B_i(v)|\right] \\
&\leq 2n + \sum_{i=0}^{t} \mathbf{E}\left[\sum_{v \in V_i} |B_i(v)|\right] \\
&\leq 2n + \sum_{i=0}^{t} \max(n^{1+1/k}, n \cdot (75 \log n)^4)/b_i \\
&\leq 2n + \sum_{i=0}^{t} \max(n^{1+1/k}, n \cdot (75 \log n)^4)/(2^i \cdot b_0) \\
&\leq 2n + 2 \cdot \max(n^{1+1/(2k)}, n \cdot (75 \log n)^2),
\end{aligned}$$

where the first inequality follows from line 5 and line 6 of Algorithm 36, the second inequality follows from Equation (6.7), the third inequality follows from Lemma 6.2.10, the forth inequality follows from $b_i > b_0 \cdot 2^i$.

The only two differences between Algorithm 35 and Algorithm 36 are as follows. The first difference is that we remove line 7 and line 11 from Algorithm 35. This change does not affect $\mathrm{dist}_{G'}(u, v)$ for any $u, v \in V$ because of Observation 6.2.17. The second difference is that we contract $v^{(0)}, v^{(1)}, \cdots$, to vertex v. We can do this operation because we have $w'(v^{(i-1)}, v^{(i)}) = 0$ for any $v^{(i-1)}, v^{(i)}$ in Algorithm 35. Then the statement follows directly from Lemma 6.2.13 □

6.3 Uncapacitated minimum cost flow

Given an undirected graph $G = (V, E, w)$ with $|V| = n$ vertices and $|E| = m$ edges, the vertex-edge incidence matrix $A \in \mathbb{R}^{n \times m}$ is defined as the following:

$$\forall i \in [n], j \in [m], A_{i,j} = \begin{cases} 1 & \{i, v\} \in E \text{ is the } j\text{-th edge of } G \text{ and } i < v, \\ -1 & \{i, v\} \in E \text{ is the } j\text{-th edge of } G \text{ and } i > v, \\ 0 & \text{Otherwise.} \end{cases}$$

The weight matrix $W \in \mathbb{R}^{m\times m}$ is a diagonal matrix. The i-th diagonal entry of W is $w(e)$, where $e \in E$ is the i-th edge. Given a demand vector $b \in \mathbb{R}^n$ with $\mathbf{1}_n^\top b = 0$, i.e., $\sum_{i=1}^n b_i = 0$, the uncapacitated minimum cost flow (transshipment) problem is to solve the following problem:

$$\begin{aligned} &\min_{f\in\mathbb{R}^m} \|Wf\|_1 \\ s.t.\ \ &Af = b. \end{aligned}$$

If b only has two non-zero entries $b_i = 1$ and $b_j = -1$, then the optimal cost is the length of the shortest path between vertex i and vertex j. Without loss of generality, we can suppose that each edge has positive weight. Otherwise, we can contract the edges with weight 0, and the contraction will not affect the value of the solution. Let $x = Wf$, then the problem becomes

$$\begin{aligned} &\min_{x\in\mathbb{R}^m} \|x\|_1 \\ s.t.\ \ &AW^{-1}x = b. \end{aligned} \tag{6.8}$$

In this section, we will focus on finding a $(1+\epsilon)$-approximation to problem (6.8).

6.3.1 Sherman's framework

Before we present our algorithm, let us review Sherman's algorithm [46], and completely open his black box.

Definition 6.3.1 (ℓ_1 Non-linear condition number)**.** *Given a matrix $B \in \mathbb{R}^{r\times m}$, the ℓ_1 non-linear condition number of B is defined as*

$$\kappa(B) = \inf_S \|B\|_{1\to 1} \cdot \sup_{x\in\mathbb{R}^m : Bx\neq 0} \frac{\|S(Bx)\|_1}{\|Bx\|_1},$$

where the range of $S : \mathbb{R}^r \to \mathbb{R}^m$ is over all maps such that $\forall x \in \mathbb{R}^m$, $B \cdot S(Bx) = Bx$.

By above definition, an alternative way to define $\kappa(B)$ is as the following:

$$\kappa(B) = \|B\|_{1\to 1} \cdot \max_{g\in\{y\in\mathbb{R}^r | y=Bx, x\in\mathbb{R}^m\}\setminus\{0\}} \min_{x:Bx=g} \frac{\|x\|_1}{\|g\|_1}.$$

Definition 6.3.2 ((α,β)-Solution)**.** *Given a matrix $B \in \mathbb{R}^{r\times m}$ and a vector $g \in \{y \in \mathbb{R}^r \mid y = Bx, x \in \mathbb{R}^m\}$, let $x^* = \arg\min_{x:Bx=g} \|x\|_1$. If $\|x\|_1 \leq \alpha\|x^*\|_1$ and $\|Bx-g\|_1 \leq \beta\|B\|_{1\to 1}\|x^*\|_1$, then x is called an (α,β)-solution with respect to (B,g). Given a matrix $B \in \mathbb{R}^{r\times m}$, if an algorithm can output an (α,β)-solution with respect to (B,g) for any vector $g \in \{y \in \mathbb{R}^r \mid y = Bx, x \in \mathbb{R}^m\}$, then the algorithm is called an (α,β)-solver for B.*

Definition 6.3.3 (Composition of the solvers)**.** *Suppose F_1 is an (α_1,β_1)-solver for $B \in \mathbb{R}^{r\times m}$ and F_2 is an (α_2,β_2)-solver for B. For any input vector $g \in \{y \in \mathbb{R}^r \mid y = Bx, x \in \mathbb{R}^m\}$, the composition $F_2 \circ F_1$ firstly runs F_1 to obtain an (α_1,β_1)-solution $x \in \mathbb{R}^m$ with respect to (B,g), then runs F_2 to obtain an (α_2,β_2)-solution $x' \in \mathbb{R}^m$ with respect to $(B, g-Bx)$, and finally outputs $x + x'$.*

Lemma 6.3.4 ([46])**.** *Suppose F_1 is an $(\alpha_1,\beta_1/\kappa)$-solver for $B \in \mathbb{R}^{r\times m}$ and F_2 is an $(\alpha_2,\beta_2/\kappa)$-solver for B, where κ is the ℓ_1 non-linear condition number of B, i.e., $\kappa = \kappa(B)$. Then $F_2 \circ F_1$ is an $(\alpha_1 + \alpha_2\beta_1, \beta_1\beta_2/\kappa)$-solver for B.*

Corollary 6.3.5 ([46])**.** *Let $\epsilon \in (0, 0.5)$. Suppose F is an $(1+\epsilon, \epsilon/\kappa)$ solver for $B \in \mathbb{R}^{r\times m}$, where κ is the ℓ_1 non-linear condition number of B, i.e., $\kappa = \kappa(B)$. Define $F^1 = F$, and $F^t = F^{t-1} \circ F$. Then F^t is an $(1+4\epsilon, \epsilon^t/\kappa)$ solver.*

Corollary 6.3.6 ([46])**.** *Let $\epsilon \in (0, 0.5), t, M \in \mathbb{R}_{\geq 0}$. Suppose F_1 is an $(1+4\epsilon, \epsilon^t/\kappa)$-solver for $B \in \mathbb{R}^{r\times m}$, and F_2 is an $(M, 0)$-solver for B, where $\kappa = \kappa(B)$. Then $F_2 \circ F_1$ is an $(1+4\epsilon+M\epsilon^t, 0)$-solver for B.*

Let us come back to the minimum cost flow problem, problem (6.8). One observation is that if a matrix $P \in \mathbb{R}^{r\times m}$ has full column rank, then $PAW^{-1}x = Pb \Leftrightarrow AW^{-1}x = b$. So, instead of solving Equation (6.8) directly, we can design a matrix $P \in \mathbb{R}^{r\times m}$ with full column rank, and try to

solve

$$\min_{x\in\mathbb{R}^m} \|x\|_1 \tag{6.9}$$
$$s.t.\ \ PAW^{-1}x = Pb.$$

Notice that since P has full column rank, problem (6.9) is exactly the same as problem (6.8). Although an $(\alpha, 0)$-solver for PAW^{-1} is also an $(\alpha, 0)$-solver for AW^{-1}, an (α, β)-solver for PAW^{-1} may not be an (α, β)-solver for AW^{-1} for $\beta > 0$. As shown in [46], if $\kappa(PAW^{-1})$ is smaller, then it is much easier to design a $(1+\epsilon, \epsilon/\kappa(PAW^{-1}))$-solver for PAW^{-1}. If $\kappa(PAW^{-1})$ is small, then we say P is a good preconditioner for AW^{-1}. Before we discuss how to construct P, let us assume $\kappa(PAW^{-1}) \leq \kappa$, and review how to solve problem (6.9).

As introduced by [46], there is a simple $(n, 0)$-solver to problem (6.9).

Algorithm 37 An $(n, 0)$-Solver

procedure MSTROUTING($G = (V, E, w), b \in \mathbb{R}^n$)
    Output: $f \in \mathbb{R}^{|E|}$
    Compute a minimum spanning tree $T = (V, E', w)$ of G.
    Choose an arbitrary vertex as the root of T. Initialize $f \in \mathbb{R}^m$.
    Consider the i-th edge $\{u, v\} \in E$ $(u < v)$. If $\{u, v\} \notin E'$, set $f_i \leftarrow 0$. Otherwise, if u is the parent of v, set $f_i \leftarrow -\sum_{z \text{ is in the subtree of } v} b_z$; otherwise, set $f_i \leftarrow \sum_{z \text{ is in the subtree of } u} b_z$.
    Return f.
end procedure

Lemma 6.3.7 ([46])**.** *Given a connected undirected weighted graph $G = (V, E, w)$, let $A \in \mathbb{R}^{n\times m}$ be the corresponding vertex-edge incidence matrix, and let $W \in \mathbb{R}^{m\times m}$ be the corresponding diagonal weight matrix, where $n = |V|, m = |E|$. For any demand vector $b \in \mathbb{R}^n$ with $\mathbf{1}_n^\top b = 0$, the output $f \in \mathbb{R}^m$ of* MSTROUTING(G, b) *(Algorithm 37) satisfies $Af = b$ and $\|Wf\|_1 \leq n \cdot \min_{f':Af'=b} \|Wf'\|_1$.*

By above lemma, if we set $x = Wf$, we have $PAW^{-1}x = Pb$, and $\|x\|_1 \leq n \cdot \min_{x':PAW^{-1}x'=Pb} \|x'\|_1$. Thus, x is an $(n, 0)$-solution to problem (6.9). Suppose $\epsilon < 0.5$. By Corollary 6.3.6, if we have a $(1+4\epsilon, \epsilon^{1+\log n}/\kappa)$ solver for PAW^{-1}, then together with Lemma 6.3.7, we can obtain a $(1+5\epsilon, 0)$-solver for PAW^{-1}, and thus we can finally find a $(1+5\epsilon)$ approximation to problem (6.8). If we

have a $(1+\epsilon, \epsilon/\kappa)$-solver for PAW^{-1}, then according to Corollary 6.3.5, we can apply $(1+\epsilon, \epsilon/\kappa)$-solver $1+\log n$ times to obtain a $(1+4\epsilon, \epsilon^{1+\log n}/\kappa)$-solver. It suffices to design a $(1+\epsilon, \epsilon/\kappa)$ solver for PAW^{-1}.

A $(1+\epsilon, \epsilon/\kappa)$-solver

In this section, we will have a detailed discussion of how [46, 67] used multiplicative weights update algorithm [95] to find a $(1+\epsilon, \epsilon/\kappa)$-solution with respect to (PAW^{-1}, Pb), where $\kappa \geq \kappa(PAW^{-1})$ is an upper bound of the condition number (see Definition 6.3.1) of PAW^{-1}, and $\epsilon \in (0, 0.5)$ is an arbitrary real number.

Let $x^* = \arg\min_{x: PAW^{-1}x = Pb} \|x\|_1$. We have

$$\frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}} \leq \|x^*\|_1 \leq \kappa \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}},$$

where the lower bound of $\|x^*\|_1$ follows from that $PAW^{-1}x^* = Pb$ and the definition of the operator ℓ_1 norm, and the upper bound of $\|x^*\|_1$ follows from the definition of condition number (see Definition 6.3.1) and thus $\|PAW^{-1}\|_{1\to 1} \cdot \frac{\|x^*\|_1}{\|Pb\|_1} \leq \kappa(PAW^{-1}) \leq \kappa$. Then, we can reduce the optimization problem to a feasibility problem. We want to binary search $s \in \{1, 1+\epsilon, (1+\epsilon)^2, \cdots, (1+\epsilon)^{\lceil \log_{1+\epsilon} \kappa \rceil}\}$, and want to find s such that $s \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}} \leq (1+\epsilon)\|x^*\|_1$ and find $x \in \mathbb{R}^m$ which satisfies $\|x\|_1 \leq s \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}}$ and $\|PAW^{-1}x - Pb\|_1 \leq \frac{\epsilon}{2\kappa} \cdot \|PAW^{-1}\|_{1\to 1} \cdot s \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}}$. The binary search will takes $O(\log(\log_{1+\epsilon} \kappa))$ rounds.

Now the problem becomes the following feasibility problem: given $s \geq 1$, either find $x \in \mathbb{R}^m$ such that

$$\|x\|_1 \leq s \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}} \quad \text{and} \quad \|PAW^{-1}x - Pb\|_1 \leq \frac{\epsilon}{2\kappa} \cdot \|PAW^{-1}\|_{1\to 1} \cdot s \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}},$$

or find a certificate such that

$$\|x\|_1 \leq s \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}} \quad \text{and} \quad PAW^{-1}x = Pb$$

is not feasible. Let $x' = x \cdot \frac{\|PAW^{-1}\|_{1\to 1}}{\|Pb\|_1} \cdot \frac{1}{s}$. Then we have the following equivalent feasibility problem: given $s \geq 1$, either find $x' \in \mathbb{R}^m$ such that

$$\|x'\|_1 \leq 1 \quad \text{and} \quad \left\| \frac{PAW^{-1}}{\|PAW^{-1}\|_{1\to 1}} x' - \frac{1}{s} \cdot \frac{Pb}{\|Pb\|_1} \right\|_1 \leq \frac{\epsilon}{2\kappa} \tag{6.10}$$

or find a certificate such that

$$\|x'\|_1 \leq 1 \quad \text{and} \quad \frac{PAW^{-1}}{\|PAW^{-1}\|_{1\to 1}} x' = \frac{1}{s} \cdot \frac{Pb}{\|Pb\|_1} \tag{6.11}$$

is not feasible.

Next, we will show how to use multiplicative weights update algorithm [95, 46, 67] to solve problem (6.10)-(6.11).

Algorithm 38 Solving the Feasibility Problem

procedure MWU($P \in \mathbb{R}^{r\times n}, A \in \mathbb{R}^{n\times m}, W \in \mathbb{R}^{m\times m}, b \in \mathbb{R}^n, s \geq 1, \epsilon \in (0, 0.5), \kappa \geq 1$)
    Output: $x' \in \mathbb{R}^m$
    Initialize weights: $\forall i \in [m], \psi_1^+(i) \leftarrow 1, \psi_1^-(i) \leftarrow 1$.
    Initialize $T \leftarrow \frac{64\kappa^2 \ln(2m)}{\epsilon^2}, \eta \leftarrow \frac{\epsilon}{8\kappa}, B \in \mathbb{R}^{n\times 2m}$:

$$B \leftarrow \left(\frac{AW^{-1}}{\|PAW^{-1}\|_{1\to 1}} - \frac{1}{s} \cdot \frac{b \cdot \mathbf{1}_m^\top}{\|Pb\|_1} \quad -\frac{AW^{-1}}{\|PAW^{-1}\|_{1\to 1}} - \frac{1}{s} \cdot \frac{b \cdot \mathbf{1}_m^\top}{\|Pb\|_1} \right).$$

    for $t = 1 \to T$ **do**
        $\Psi_t \leftarrow \sum_{i=1}^m \psi_t^+(i) + \sum_{i=1}^m \psi_t^-(i)$.
        For $i \in [m]$, $p_t^+(i) \leftarrow \psi_t^+(i)/\Psi_t, p_t^-(i) \leftarrow \psi_t^-(i)/\Psi_t$.
        Set $p_t \in \mathbb{R}^{2m}$ s.t. $\forall i \in [m]$, the i-th entry of p_t is $p_t^+(i)$, and the $(i+m)$-th entry of p_t is $p_t^-(i)$.
        If $\|PBp_t\|_1 \leq \frac{\epsilon}{2\kappa}$, return $x' \in \mathbb{R}^m$ such that $\forall i \in [m], x_i' = p_t^+(i) - p_t^-(i)$.
        Otherwise, set $y_t \in \{+1, -1\}^r$ such that $\forall i \in [r], (y_t)_i = \operatorname{sgn}\left((PBp_t)_i\right)$.
        For $i \in [m]$, $\phi_t^+(i) \leftarrow y_t^\top PB_i/2, \phi_t^-(i) \leftarrow y_t^\top PB_{i+m}/2$.
        For $i \in [m]$, $\psi_{t+1}^+(i) \leftarrow \psi_t^+(i) \cdot \left(1 - \eta\phi_t^+(i)\right), \psi_{t+1}^-(i) \leftarrow \psi_t^-(i) \cdot \left(1 - \eta\phi_t^-(i)\right)$.
    end for
    Return FAIL.
end procedure

Lemma 6.3.8 ([46, 67])**.** *Consider $P \in \mathbb{R}^{r\times n}, A \in \mathbb{R}^{n\times m}, W \in \mathbb{R}^{m\times m}, b \in \mathbb{R}^n, s \geq 1, \epsilon \in (0, 0.5), \kappa \geq 1$. $\mathrm{MWU}(P, A, W, b, s, \epsilon, \kappa)$ (Algorithm 38) takes $T = O(\kappa^2\epsilon^{-2}\log m)$ iterations. If $\mathrm{MWU}(P, A, W, b, s, \epsilon, \kappa)$ does not return FAIL, the output $x' \in \mathbb{R}^m$ satisfies Equation* (6.10)*. Otherwise, $\bar{y} = \frac{1}{T}\sum_{t=1}^T y_t$ is a*

certificate that Equation (6.11) *is not feasible. In particular,*

$$\forall j \in [m], \quad \frac{1}{s} \cdot \frac{\bar{y}^\top Pb}{\|Pb\|_1} < \frac{\bar{y}^\top (PAW^{-1})_j}{\|PAW^{-1}\|_{1\to 1}}, \quad \frac{1}{s} \cdot \frac{\bar{y}^\top Pb}{\|Pb\|_1} < -\frac{\bar{y}^\top (PAW^{-1})_j}{\|PAW^{-1}\|_{1\to 1}}.$$

For completeness, we put the proof here.

Proof. Let us firstly consider the case that $\mathrm{MWU}(P, A, W, b, s, \epsilon, \kappa)$ does not return FAIL. By line 9, we have $\|PBp_t\|_1 \leq \frac{\epsilon}{2\kappa}$, i.e.,

$$\left\| \sum_{i=1}^{m} \left(\frac{(PAW^{-1})_i}{\|PAW^{-1}\|_{1\to 1}} \cdot (p_t^+(i) - p_t^-(i)) - \frac{1}{s} \cdot \frac{Pb}{\|Pb\|_1} \cdot (p_t^+(i) + p_t^-(i)) \right) \right\|_1 \leq \frac{\epsilon}{2\kappa}.$$

Since $\sum_{i=1}^{m} (p_t^+(i) + p_t^-(i)) = 1$ and $x_i' = p_t^+(i) - p_t^-(i)$, we have:

$$\left\| \frac{PAW^{-1}}{\|PAW\|_{1\to 1}} x' - \frac{1}{s} \cdot \frac{Pb}{\|Pb\|_1} \right\|_1 \leq \frac{\epsilon}{2\kappa}.$$

Furthermore, because $\forall i \in [m], p_t^+(i), p_t^-(i) \geq 0$, we have $\|x'\|_1 \leq 1$. Thus, x' satisfies Equation (6.10).

Let us consider the case when $\mathrm{MWU}(P, A, W, b, s, \epsilon, \kappa)$ outputs FAIL. For $i \in [m], t \in [T]$, we have

$$\begin{aligned}
|\phi_t^+(i)| &\leq \|y_t\|_\infty \|PB_i\|_1 / 2 \\
&= \left\| \frac{(PAW^{-1})_i}{\|PAW^{-1}\|_{1\to 1}} - \frac{1}{s} \cdot \frac{Pb}{\|Pb\|_1} \cdot \right\|_1 / 2 \\
&\leq \left\| \frac{(PAW^{-1})_i}{\|PAW^{-1}\|_{1\to 1}} \right\|_1 / 2 + \left\| \frac{1}{s} \cdot \frac{Pb}{\|Pb\|_1} \cdot \right\|_1 / 2 \\
&\leq 1,
\end{aligned}$$

where the first step follows from Höder's inequality, the second step follows from the construction of B, the third step follows from triangle inequality, and the last step follows from $\|(PAW^{-1})_i\|_1 \leq \|PAW^{-1}\|_{1\to 1}$, $\|Pb/\|Pb\|_1\|_1 = 1$ and $s \geq 1$. Similarly, we also have $|\phi_t^-(i)| \leq 1$. By Theorem 2.1

of [95]:

$$\forall j \in [m], \sum_{t=1}^{T}\sum_{i=1}^{m}(p_t^+(i)\phi_t^+(i) + p_t^-(i)\phi_t^-(i)) \leq \sum_{t=1}^{T}\phi_t^+(j) + \eta\sum_{t=1}^{T}|\phi_t^+(j)| + \frac{\ln(2m)}{\eta},$$
$$\forall j \in [m], \sum_{t=1}^{T}\sum_{i=1}^{m}(p_t^+(i)\phi_t^+(i) + p_t^-(i)\phi_t^-(i)) \leq \sum_{t=1}^{T}\phi_t^-(j) + \eta\sum_{t=1}^{T}|\phi_t^-(j)| + \frac{\ln(2m)}{\eta}.$$

By the construction of $p_t^+, p_t^-, \phi_t^+, \phi_t^-$,

$$\sum_{t=1}^{T}\sum_{i=1}^{m}(p_t^+(i)\phi_t^+(i) + p_t^-(i)\phi_t^-(i)) = \sum_{t=1}^{T} y_t^\top PBp_t/2 > T\cdot\frac{\epsilon}{4\kappa},$$

where the inequality follows from $\forall l \in [r], (y_t)_l = \operatorname{sgn}((PBp_t)_i)$ and thus $y_t^\top PBp_t = \|PBp_t\|_1 > \frac{\epsilon}{2\kappa}$. Thus,

$$\forall j \in [m], T\cdot\frac{\epsilon}{4\kappa} < \sum_{t=1}^{T}\phi_t^+(j) + \eta T + \frac{\ln(2m)}{\eta}, \tag{6.12}$$
$$\forall j \in [m], T\cdot\frac{\epsilon}{4\kappa} < \sum_{t=1}^{T}\phi_t^-(j) + \eta T + \frac{\ln(2m)}{\eta}. \tag{6.13}$$

Let $\bar{y} = \frac{1}{T}\sum_{t=1}^{T} y_t$, then

$$\forall j \in [m], \sum_{t=1}^{T}\phi_t^+(j) = T\cdot\bar{y}^\top\left(\frac{(PAW^{-1})_j}{\|PAW^{-1}\|_{1\to1}} - \frac{1}{s}\cdot\frac{Pb}{\|Pb\|_1}\right),$$
$$\forall j \in [m], \sum_{t=1}^{T}\phi_t^-(j) = T\cdot\bar{y}^\top\left(-\frac{(PAW^{-1})_j}{\|PAW^{-1}\|_{1\to1}} - \frac{1}{s}\cdot\frac{Pb}{\|Pb\|_1}\right).$$

Recall that $\eta = \frac{\epsilon}{8\kappa}, T = \frac{64\kappa^2\ln(2m)}{\epsilon^2}$. Thus, together with Equation (6.12) and Equation (6.13), we have:

$$\forall j \in [m], 0 < \bar{y}^\top\left(\frac{(PAW^{-1})_j}{\|PAW^{-1}\|_{1\to1}} - \frac{1}{s}\cdot\frac{Pb}{\|Pb\|_1}\right),$$

$$\forall j \in [m], 0 < \bar{y}^\top \left(-\frac{(PAW^{-1})_j}{\|PAW^{-1}\|_{1\to 1}} - \frac{1}{s} \cdot \frac{Pb}{\|Pb\|_1} \right).$$

For any $x' \in \mathbb{R}^m$ with $\|x'\|_1 \leq 1$, we can always find $x'^+, x'^- \in \mathbb{R}^m$ such that $x'^+, x'^- \geq 0, x' = x'^+ - x'^-$, and $\sum_{i=1}^m (x_i'^+ + x_i'^-) = 1$. If x' satisfies Equation (6.11), then

$$\begin{aligned} 0 &= \bar{y}^\top \left(\frac{PAW^{-1}}{\|PAW^{-1}\|_{1\to 1}} (x'^+ - x'^-) - \frac{1}{s} \cdot \frac{Pb}{\|Pb\|_1} \right) \\ &= \sum_{j=1}^m \left(\bar{y}^\top \left(\frac{(PAW^{-1})_j}{\|PAW^{-1}\|_{1\to 1}} - \frac{1}{s} \cdot \frac{Pb}{\|Pb\|_1} \right) \cdot x_j'^+ + \bar{y}^\top \left(-\frac{(PAW^{-1})_j}{\|PAW^{-1}\|_{1\to 1}} - \frac{1}{s} \cdot \frac{Pb}{\|Pb\|_1} \right) \cdot x_j'^- \right) \\ &> 0 \end{aligned}$$

which leads to a contradiction. □

6.3.2 Preconditioner construction

As discussed in the previous section, if we can find a good preconditioner such that $\kappa(PAW^{-1})$ is small, then we can use a small number of iterations to compute a good solution. Before we describe how to choose a good preconditioner, let us introduce the following Lemma.

Lemma 6.3.9 ([46, 67])**.** *Given $P \in \mathbb{R}^{r\times n}$ with full column rank, $A \in \mathbb{R}^{n\times m}, W \in \mathbb{R}^{m\times m}$, if $\forall b \in \{y \in \mathbb{R}^n \mid y = AW^{-1}x, x \in \mathbb{R}^m\}$,*

$$\|x^*\|_1 \leq \|Pb\|_1 \leq \gamma \|x^*\|_1,$$

where $x^* = \arg\min_{x\in\mathbb{R}^m : AW^{-1}x = b} \|x\|_1$, then $\kappa(PAW^{-1}) \leq \gamma$.

Proof. For any $x \in \mathbb{R}^m$, $\|PAW^{-1}x\|_1 \leq \gamma\|x\|_1$. Thus, $\|PAW^{-1}\|_{1\to 1} \leq \gamma$. By Definition 6.3.1, we have:

$$\kappa(PAW^{-1}) = \|PAW^{-1}\|_{1\to 1} \cdot \max_{b\in\{y\in\mathbb{R}^n \mid y = AW^{-1}x, x\in\mathbb{R}^m\} : Pb\neq 0} \min_{x\in\mathbb{R}^m : PAW^{-1}x = Pb} \frac{\|x\|_1}{\|Pb\|_1} \leq \gamma,$$

where the inequality follows from $\|PAW^{-1}\|_{1\to 1} \leq \gamma$, P has full column rank and $\forall b \in \{y \in \mathbb{R}^n \mid y = AW^{-1}x, x \in \mathbb{R}^m\}$,

$$\min_{x\in\mathbb{R}^m : AW^{-1}x=b} \|x\|_1 \leq \|Pb\|_1.$$

□

By above lemma, our goal is to find a linear operator P such that for any demand vector b, $\|Pb\|_1$ can approximate the minimum cost flow with demand vector b very well. Instead of using Sherman's original lattice algorithm, we propose to use randomly shifted grids based algorithm [96].

Embedding minimum cost flow into ℓ_1 via randomly shifted grids

In this section, we review the embedding method of [96] and describe how to construct the preconditioner. Suppose we have a mapping $\varphi : V \to [\Delta]^d$ such that $\forall u, v \in V$,

$$\mathrm{dist}_G(u, v) \leq \|\varphi(u) - \varphi(v)\|_1 \leq \alpha \cdot \mathrm{dist}_G(u, v).$$

We can reduce estimating the minimum cost flow on G to approximating the cost of the geometric transportation problem. The geometric transportation problem is also called Earth Mover's Distance (EMD) problem. In particular, it is the following minimization problem:

$$\min_{\pi : V\times V \to \mathbb{R}_{\geq 0}} \sum_{(u,v)\in V\times V} \pi(u, v) \cdot \|\varphi(u) - \varphi(v)\|_1 \tag{6.14}$$
$$s.t.\ \forall u \in V, \sum_{v\in V} \pi(u, v) - \sum_{v\in V} \pi(v, u) = b_u.$$

It is obvious that if we can obtain a β-approximation to the optimal cost of (6.14), we can obtain an $\alpha\beta$-approximation to the cost of original minimum cost flow problem on G.

For a sequential algorithm, the such embedding φ can be obtained by Bourgain's Embedding.

Lemma 6.3.10 (Bourgain's Embedding [55])**.** *Given an undirected graph $G = (V, E, w)$ with $|V| = n$ vertices and $|E| = m$ edges, there is a randomized algorithm which can output a mapping $\varphi : V \to [\Delta]^d$ for $d = O(\log^2 n)$ with probability 0.99 in $O(m \log^2 n)$ time, such that*

$$\forall u, v \in V, \mathrm{dist}_G(u, v) \leq \|\varphi(u) - \varphi(v)\|_1 \leq O(\log n) \cdot \mathrm{dist}_G(u, v),$$

where $\Delta \leq \sum_{e \in E} w(e)$.

In the remaining of this section, we focus on approximating (6.14). Without loss of generality, we suppose Δ is a power of 2. Let $L = 1 + \log \Delta$. We create L levels grids $G_0, G_1, \cdots, G_{L-1}$, where G_i partitions $[2\Delta]^d$ into disjoint cells with side length 2^i. In particular, $\forall i \in \{0, 1 \cdots, L-1\}$, the i-th level grid G_i is:

$$\left\{ C \;\middle|\; C = \{a_1, \cdots, a_1 + 2^i - 1\} \times \cdots \times \{a_d, \cdots, a_d + 2^i - 1\}, \forall j \in [d], a_j \bmod 2^i = 1, a_j \in [2\Delta] \right\}.$$

Instead of shifting the gird, we shift the points. For each dimension, we can use the same shift value τ [97]. Let τ be a random variable with uniform distribution over $[\Delta]$. We can construct a vector $h \in \mathbb{R}^{\sum_{i=0}^{L-1} |G_i|}$ with one entry per cell in $G_0 \cup G_1 \cup \cdots \cup G_{L-1}$. Let $h_{(i,C)}$ correspond to the cell $C \in G_i$. For each $i \in \{0, 1, \cdots, L-1\}$ and each cell $C \in G_i$, we set $h_{(i,C)}$ as:

$$h_{(i,C)} = d \cdot 2^i \cdot \sum_{v \in V : \varphi(v) + \tau \cdot \mathbf{1}_d \in C} b_v.$$

Let $\mathrm{OPT}_{\mathrm{EMD}}(b)$ denote the optimal solution of the EMD problem (6.14). As shown by [96], $\|h\|_1$ is a good approximation to $\mathrm{OPT}_{\mathrm{EMD}}(b)$.

Lemma 6.3.11. *Let $h \in \mathbb{R}^{\sum_{i=0}^{L-1} |G_i|}$ be constructed as above. Then,*

1. $\mathbf{E}_\tau[\|h\|_1] \leq 2Ld \cdot \mathrm{OPT}_{EMD}$,
2. $\|h\|_1 \geq \mathrm{OPT}_{\mathrm{EMD}}(b)$.

Proof. Consider the upper bound. Let $\pi^* : V \times V \to \mathbb{R}_{\geq 0}$ be the optimal solution of problem (6.14).

$$\begin{aligned}
\mathop{\mathbf{E}}_{\tau}[\|h\|_1] &\leq \sum_{i=0}^{L-1} \sum_{(u,v)\in V\times V} 2 \cdot d \cdot 2^i \cdot \pi^*(u,v) \cdot \Pr_{\tau}[\varphi(u) + \tau \cdot \mathbf{1}_d \text{ and } \varphi(v) + \tau \cdot \mathbf{1}_d \text{ are in different cells of } G_i] \\
&\leq \sum_{i=0}^{L-1} \sum_{(u,v)\in V\times V} 2 \cdot d \cdot 2^i \cdot \pi^*(u,v) \cdot \sum_{j=1}^{d} \frac{|\varphi(u)_j - \varphi(v)_j|}{2^i} \\
&\leq 2Ld \cdot \sum_{(u,v)\in V\times V} \pi^*(u,v) \|\varphi(u) - \varphi(v)\|_1 \\
&= 2Ld \cdot \mathrm{OPT}_{\mathrm{EMD}}(b),
\end{aligned}$$

where the second step follows from union bound on all dimensions.

Consider the lower bound. We can build a tree with one node per cell in $G_0 \cup G_1 \cup \cdots \cup G_{L-1}$. For a cell $C \in G_i$, there is a unique cell $C' \in G_{i+1}$ such that $C \subset C'$. We connect the nodes corresponding to C and C' with an edge of which weight is $d \cdot 2^i$. For $u, v \in V$, there are two cells $C_1, C_2 \in G_0$ such that $\varphi(u) + \tau \cdot \mathbf{1}_d \in C_1$ and $\varphi(v) + \tau \cdot \mathbf{1}_d \in C_2$. The distance between two nodes corresponding to C_1, C_2 on the tree is at least $\|\varphi(u) - \varphi(v)\|_1$. The cost of the minimum cost flow on the such tree is

$$\sum_{i=0}^{L-1} \sum_{C\in G_i} d \cdot 2^i \cdot \left| \sum_{v\in V:\varphi(v)+\tau\cdot\mathbf{1}_d\in C} b_v \right| = \|h\|_1.$$

Thus, $\|h\|_1 \geq \mathrm{OPT}_{\mathrm{EMD}}(b)$. □

An observation is that since each cell in G_i has side length 2^i, shifting each point by $\tau \cdot \mathbf{1}_d$ is equivalent to shifting each point by $(\tau \bmod 2^i) \cdot \mathbf{1}_d$ for the cells in G_i. Thus, if we modify the construction of h as the following:

$$\forall i \in \{0, 1, \cdots, L-1\}, C \in G_i, h_{(i,C)} = d \cdot 2^i \cdot \sum_{v\in V:\varphi(v)+(\tau \bmod 2^i)\cdot\mathbf{1}_d\in C} b_v,$$

Lemma 6.3.11 still holds. Next, we describe how to construct $h' \in \mathbb{R}^{\sum_{i=0}^{L-1} 2^i |G_i|}$. The entry $h'_{(i,C,\tau)}$

corresponds to the cell $C \in G_i$ and the shift value τ. For each $i \in \{0, 1, \cdots, L-1\}$, each cell $C \in G_i$ and each shift value $\tau \in [2^i]$, we set $h'_{(i,C,\tau)}$ as:

$$h'_{(i,C,\tau)} = \frac{1}{2^i} \cdot d \cdot 2^i \cdot \sum_{v \in V : \varphi(v) + \tau \cdot \mathbf{1}_d \in C} b_v = d \cdot \sum_{v \in V : \varphi(v) + \tau \cdot \mathbf{1}_d \in C} b_v.$$

It is clear that $\|h'\|_1 = \mathbf{E}[\|h\|_1]$. By Lemma 6.3.11, we have

$$\mathrm{OPT}_{\mathrm{EMD}}(b) \leq \|h'\|_1 \leq 2Ld \cdot \mathrm{OPT}_{\mathrm{EMD}}(b).$$

Observe that h' can be written as a linear map of b, i.e., $h' = P'b$, where $P' \in \mathbb{R}^{(\sum_{i=0}^{L-1} 2^i |G_i|) \times n}$. Each row of P' is indexed by a tuple (i, C, τ) for $i \in \{0, 1, \cdots, L-1\}, C \in G_i$ and $\tau \in [2^i]$, and each column of P' is indexed by a vertex $v \in V$. For $i \in \{0, 1, \cdots, L-1\}, C \in G_i, \tau \in [2^i], v \in V$,

$$P'_{(i,C,\tau),v} = \begin{cases} d & \varphi(v) + \tau \cdot \mathbf{1}_d \in C, \\ 0 & \text{Otherwise.} \end{cases}$$

Consider $i = 0, \tau = 1$, $\forall v \in V$, there is a unique cell $C \in G_0$ which contains $\varphi(v) + \mathbf{1}_d$. Thus, P' has full column rank. According to Lemma 6.3.9, since $\forall b \in \{y \in \mathbb{R}^n \mid y = AW^{-1}x, x \in \mathbb{R}^m\}$,

$$\min_{x \in \mathbb{R}^m : AW^{-1}x = b} \|x\|_1 \leq \mathrm{OPT}_{\mathrm{EMD}}(b) \leq \|P'b\|_1 \leq 2Ld \cdot \mathrm{OPT}_{\mathrm{EMD}}(b) \leq 2Ld\alpha \cdot \min_{x \in \mathbb{R}^m : AW^{-1}x = b} \|x\|_1,$$

we have $\kappa(P'AW^{-1}) \leq 2Ld\alpha$. However, since the size of P' is too large, we cannot apply P' directly in Algorithm 38, and thus it is unclear how to construct a $(1 + \epsilon, \epsilon/\kappa(P'AW^{-1}))$-solver for $P'AW^{-1}$.

6.3.3 Fast operations for the preconditioner

One of our main contributions is to develop several fast operations for P' such that we can implement Algorithm 38 efficiently.

Removing useless cells. The first observation is that though P' has a large number of rows, most rows of P' are zero. Thus, we can remove them. Precisely, for each $i \in \{0, 1, \cdots, L-1\}$, let $C_i = \{C \in G_i \mid \exists v \in V, \tau \in [2^i],\ s.t.\ \varphi(v) + \tau \cdot \mathbf{1}_d \in C\}$. Then we can set $P \in \mathbb{R}^{(\sum_{i=0}^{L-1} 2^i |C_i|) \times n}$ such that $\forall i \in \{0, 1, \cdots, L-1\}, C \in C_i, \tau \in [2^i], v \in V$,

$$P_{(i,C,\tau),v} = \begin{cases} d & \varphi(v) + \tau \cdot \mathbf{1}_d \in C, \\ 0 & \text{Otherwise.} \end{cases}$$

Lemma 6.3.12. $\forall i \in \{0, 1, \cdots, L-1\}$, $|C_i| \leq n \cdot (d+1)$.

Proof. Since each cell has side length 2^i, for a dimension $j \in [d]$ and a vertex $v \in V$, $\varphi(v) + \mathbf{1}_d, \varphi(v) + 2 \cdot \mathbf{1}_d, \cdots, \varphi(v) + 2^i \cdot \mathbf{1}_d$ can cross the boundary in the j-th dimension at most once. Therefore, $\varphi(v) + \mathbf{1}_d, \varphi(v) + 2 \cdot \mathbf{1}_d, \cdots, \varphi(v) + 2^i \cdot \mathbf{1}_d$ can be in at most $d+1$ different cells in G_i. Because V has size n, we can conclude $|C_i| \leq n \cdot (d+1)$. □

By Lemma 6.3.12, we know that P has at most $2\Delta \cdot n(d+1)$ rows. This is still too large.

Compressed representation. Another observation is that, P may have many identical rows. Thus, we want to handle these rows simultaneously. To achieve this goal, we introduce a concept called compressed representation.

Definition 6.3.13 (Compressed representation of a vector)**.** *Let $I = \{([a_1, b_1], c_1), \cdots, ([a_s, b_s], c_s)\}$, where $c_i \in \mathbb{R}$, $[a_i, b_i] \subseteq [1, r]$ for some $r \in \mathbb{Z}_{\geq 1}$, and $\forall i \neq j \in [s], [a_i, b_i] \cap [a_j, c_j] = \emptyset$. Let $x \in \mathbb{R}^r$. If $\forall i \in [s], j \in [a_i, b_i], x_j = c_i$ and $\forall j \in [1, r] \setminus \bigcup_{i \in [s]} [a_i, b_i], x_j = 0$, then I is an compressed representation of x. The size of the compressed representation I is $|I| = s$.*

By above definition, the compressed representation of x is not unique.

Definition 6.3.14 (Compressed representation of a matrix)**.** *Let $I = (I_1, I_2, \cdots, I_n)$. Given a matrix $P \in \mathbb{R}^{r \times n}$, if $\forall i \in [n]$, I_i is an compressed representation of P_i, then I is called an compressed representation of P. Furthermore, the size of the compressed representation I is defined as $\sum_{i=1}^n |I_i|$.*

Algorithm 39 Computing an compressed representation of P

procedure IMPLICITP($\varphi : V \to [\Delta]^d$)
  Output: I
  $n \leftarrow |V|, L \leftarrow 1 + \log \Delta, \forall i \in \{0, 1, \cdots, L-1\}, C_i \leftarrow \emptyset$, and create grids $G_0, G_1, \cdots, G_{L-1}$.
  $\forall i \in \{0, 1, \cdots, L-1\}, v \in V, C_i \leftarrow C_i \cup \{C \in G_i \mid \exists \tau \in [2^i], \varphi(v) + \tau \cdot \mathbf{1}_d \in C\}$.
  for the i-th vertex $v \in V$ **do**
    $I_i \leftarrow \emptyset$.
    for $l \in \{0, 1, \cdots, L-1\}$ **do**
      For each $C \in C_l$ with $\exists \tau \in [2^l], \varphi(v) + \tau \cdot \mathbf{1}_d \in C$, find $\tau_1, \tau_2 \in [2^l]$ such that

$$\tau_1 = \min_{\tau \in [2^l] : \varphi(v) + \tau \cdot \mathbf{1}_d \in C} \tau, \quad \tau_2 = \max_{\tau \in [2^l] : \varphi(v) + \tau \cdot \mathbf{1}_d \in C} \tau.$$

      Suppose C is the k-th cell in C_l. $a \leftarrow (k-1)2^l + \sum_{j=0}^{l-1} 2^j |C_j|$.

$$I_i \leftarrow I_i \cup \{([a + \tau_1, a + \tau_2], d)\}.$$

    end for
  end for
  Return $I = (I_1, I_2, \cdots, I_n)$.
end procedure

Lemma 6.3.15 (Computing an compressed representation of P)**.** *Given an undirected graph $G = (V, E, w)$ with $|V| = n, |E| = m$ and a mapping $\varphi : V \to [\Delta]^d$ for some Δ, d, such that*

$$\forall u, v \in V, \mathrm{dist}_G(u, v) \leq \|\varphi(u) - \varphi(v)\|_1 \leq \alpha \cdot \mathrm{dist}_G(u, v),$$

the output $I = (I_1, I_2, \cdots, I_n)$ of IMPLICITP(φ) *(Algorithm 39) is an compressed representation of a matrix P with full column rank and $\kappa(PAW^{-1}) \leq O(\alpha L d)$, where $L = 1 + \log \Delta$, $A \in \mathbb{R}^{n \times m}$ is the vertex-incidence matrix, and $W \in \mathbb{R}^{m \times m}$ is the diagonal weight matrix. Furthermore, for $i \in [n]$, the size of I_i is at most $(d+1)L$. The running time of* IMPLICITP(φ) *is $n \cdot \mathrm{poly}(dL \log n)$.*

Proof. As discussed previously, P has full column rank and $\kappa(PAW^{-1}) \leq O(\alpha L d)$ by our construction. As discussed in the proof of Lemma 6.3.12, $\forall l \in \{0, 1, \cdots, L-1\}, v \in V$, we know that $\varphi(v) + \mathbf{1}_d, \varphi(v) + 2 \cdot \mathbf{1}_d, \cdots, \varphi(v) + 2^i \cdot \mathbf{1}_d$ can be in at most $d+1$ different cells in G_i. Thus, $\forall i \in [n], |I_i| \leq (d+1)L$. Notice that $\forall v \in V, \varphi(v)$ has dimension d, we only need to handle L levels, and sorting will only induce additional $\log n$ factors, the running time will be at most

$n \cdot \mathrm{poly}(dL \log n)$. □

Operations under compressed representations

In this section, we introduce how to implement some important operations under compressed representations.

Fact 6.3.16. *Let $I = \{([a_1, b_1], c_1), ([a_2, b_2], c_2), \cdots, ([a_s, b_s], c_s)\}$ be an compressed representation of a vector $x \in \mathbb{R}^r$. Then, $\|x\|_1 = \sum_{i=1}^{s} (b_i - a_i + 1) \cdot |c_i|$. Let $y \in \mathbb{R}^r$ be the vector satisfying $\forall i \in [r], y_i = \mathrm{sgn}(x_i)$. Then $I' = \{([a_1, b_1], \mathrm{sgn}(c_1)), \cdots, ([a_s, b_s], \mathrm{sgn}(c_s))\}$ is an compressed representation of y. Let $z = t \cdot x$, where t is a scalar. Then $I'' = \{([a_1, b_1], tc_1), \cdots, ([a_s, b_s], tc_s)\}$ is an compressed representation of a vector z. Furthermore, both $\|x\|_1$, I' and I'' can be computed in $O(s)$ time.*

Algorithm 40 Compressed Matrix-Vector Multiplication

procedure MATRIXVEC($I = (I_1, I_2, \cdots, I_n), g \in \mathbb{R}^n$)
  Output: $\widehat{I}$
  $S \leftarrow \emptyset, \widehat{I} \leftarrow \emptyset$.
  for $i \in [n] : g_i \neq 0$ **do**
    For each $([a, b], c) \in I_i$, $S \leftarrow S \cup \{(a, cg_i), (b + 1, -cg_i)\}$.
  end for
  Sort $S = \{(q_1, z_1), (q_2, z_2), \cdots, (q_k, z_k)\}$ such that $q_1 \leq q_2 \leq \cdots \leq q_k$.
  For each $j \in \{2, 3, \cdots, k\} : q_j > q_{j-1}$, $\widehat{I} \leftarrow \widehat{I} \cup \{([q_{j-1}, q_j - 1], \sum_{t:q_t<q_j} z_t)\}$.
  Return $\widehat{I}$.
end procedure

Lemma 6.3.17 (Compressed matrix-vector multiplication)**.** *Given an compressed representation $I = (I_1, I_2, \cdots, I_n)$ of a matrix $P \in \mathbb{R}^{r \times n}$ with $\forall i \in [n], |I_i| \leq s$, and a vector $g \in \mathbb{R}^n$, the output $\widehat{I}$ of* MATRIXVEC(I, g) *(Algorithm 40) is an compressed representation of Pg. Furthermore, $|\widehat{I}| \leq 2s \cdot \mathrm{nnz}(g)$, and the running time is at most $O(s\,\mathrm{nnz}(g) \cdot \log(s\,\mathrm{nnz}(g)))$.*

Proof. Consider $j \in [r]$ such that $(Pg)_j \neq 0$.

$$(Pg)_j = \sum_{i \in [n]: g_i \neq 0} P_{j,i} g_i = \sum_{i \in [n]: g_i \neq 0, \exists ([a,b], c) \in I_i, j \in [a,b]} c g_i.$$

Notice that $\forall h \in [k]$, $\sum_{t:q_t<q_h} z_t = \sum_{i\in[n]:g_i\neq 0,\exists([a,b],c)\in I_i,a\leq q_t,b\geq q_h} cg_i$. Since we can always find $h \in \{2,3,\cdots,k\}$ such that $j \in [q_{h-1}, q_h - 1]$, then for such h we have $(Pg)_j = \sum_{t:q_t<q_h} z_t$. Thus, $\widehat{I}$ is an compressed representation of Pg.

For $i \in [n] : g_i \neq 0$, we will add at most 2 elements in S for each tuple in I_i. Since each element in S can correspond to at most 1 tuple in $\widehat{I}$, we have $|\widehat{I}| \leq 2s\,\mathrm{nnz}(g)$. Sorting takes $O(|S|\log|S|)$ time, and maintaining prefix sum takes $O(|S|)$ time. Thus, total running time is at most $O(|S|\log|S|) = O(s\,\mathrm{nnz}(g)\log(s\,\mathrm{nnz}(g)))$. □

Algorithm 41 Compressed Vector-Matrix Multiplication

procedure VECTORMAT($I, I' = (I_1, I_2, \cdots, I_n)$)
  Output: $g^\top \in \mathbb{R}^n$
  $g \leftarrow (0, 0, \cdots, 0)$.
  Fill I such that $\forall j \in [r], \exists([a,b],c) \in I, j \in [a,b]$.
  Sort $I = \{([a_1,b_1],c_1),([a_2,b_2],c_2),\cdots,([a_s,b_s],c_s)\}$ such that $a_1 < a_2 < \cdots < a_s$.
  $\forall j \in [s]$, compute the prefix sum $p_j = \sum_{t=1}^{j}(b_t - a_t + 1)\cdot c_t$.
  for $i \in [n]$ **do**
    for $([a,b],c) \in I_i$ **do**
      Run binary search to find $j_1 \leq j_2$ such that $a \in [a_{j_1}, b_{j_1}], b \in [a_{j_2}, b_{j_2}]$.
      If $j_1 = j_2$, $g_i \leftarrow g_i + c \cdot c_{j_1} \cdot (b - a + 1)$.
      If $j_1 < j_2$, $g_i \leftarrow g_i + c \cdot (c_{j_1} \cdot (b_{j_1} - a + 1) + c_{j_2} \cdot (b - a_{j_2} + 1) + (p_{j_2-1} - p_{j_1}))$.
    end for
  end for
  Return $g^\top$.
end procedure

Lemma 6.3.18 (Compressed vector-matrix multiplication)**.** *Given an compressed representation I of a vector $y \in \mathbb{R}^r$ with $|I| \leq s$ and an compressed representation $I' = (I_1, I_2, \cdots, I_n)$ of a matrix $P \in \mathbb{R}^{r\times n}$ with $\forall i \in [n], |I_i| \leq s'$, the output $g^\top \in \mathbb{R}^n$ of* VECTORMAT(I, I') *(Algorithm 41) is $P^\top y$. Furthermore, the running time is $O((s + ns')\log s)$.*

Proof. Consider the i-th entry of $y^\top P$,

$$(y^\top P)_i = y^\top P_i = \sum_{([a,b],c)\in I_i} c \cdot \sum_{a\leq t\leq b} y_t.$$

By our algorithm, it is easy to show that $(y^\top P)_i = g_i$. Sorting I takes $O(s\log s)$ time. For each

$i \in [n]$, we need to take $O(s' \log s)$ time for s' times binary search. Thus, the total running time is $O((s + s'n) \log s)$. □

6.3.4 Uncapacitated minimum cost flow algorithm

Theorem 6.3.19. *Given an $\epsilon \in (0, 0.5)$, a connected n-vertex m-edge undirected graph $G = (V, E, w)$ with $w : E \to \mathbb{Z}_{\geq 0}$, and a demand vector $b \in \mathbb{R}^n$ with $\mathbf{1}_n^\top b = 0$, there is a randomized algorithm which can output an $(1 + \epsilon)$-approximate solution to the uncapacitated minimum cost flow problem in $\epsilon^{-2} m \cdot (\log n \log \Lambda)^{O(1)}$ time with probability at least 0.99, where $\Lambda = \sum_{e \in E} w(e)$.*

Proof. Let $\Lambda = \sum_{e \in E} w(e)$. Let $A \in \mathbb{R}^{n \times m}$ be the vertex-edge incidence matrix of G, and let $W \in \mathbb{R}^{m \times m}$ be the weight matrix. By Lemma 6.3.10, with 0.99 probability, we can compute a mapping $\varphi : V \to [\Delta]^d$ with $\Delta \leq \Lambda, d \leq O(\log^2 n)$ in $O(m \log^2 n)$ time such that

$$\forall u, v \in V, \mathrm{dist}_G(u, v) \leq \|\varphi(u) - \varphi(v)\|_1 \leq O(\log n) \cdot \mathrm{dist}_G(u, v).$$

By Lemma 6.3.15, we can compute an compressed representation $I = \{I_1, I_2, \cdots, I_n\}$ of P with $\forall i \in [n], |I_i| \leq O(d \log \Delta) = O(\log^2 n \log \Lambda)$. Furthermore, $\kappa(PAW^{-1}) \leq O(\log^3 n \log \Lambda)$. The running time is $n \cdot (\log n \log \Lambda)^{O(1)}$. Now, we are able to implement Algorithm 38. To compute matrix B, we need to compute $\|PAW^{-1}\|_{1 \to 1}$ and $\|Pb\|_1$. Notice that $\|PAW^{-1}\|_{1 \to 1} = \max_{i \in [m]} \|P(AW^{-1})_i\|_1$ and $(AW^{-1})_i$ only has two non-zero entries. By Lemma 6.3.17, an compressed representation of $P(AW^{-1})_i$ can be computed in $O(\log^2 n \log \Lambda \cdot (\log \log n + \log \log \Lambda))$ time, and the size of the compressed representation is at most $O(\log^2 n \log \Lambda)$. By Fact 6.3.16, $\|P(AW^{-1})_i\|_1$ can be computed in $O(\log^2 n \log \Lambda (\log \log n + \log \log \Lambda))$ time. Thus, $\|PAW^{-1}\|_{1 \to 1}$ can be computed in $m \cdot (\log n \log \Lambda)^{O(1)}$ time. By Lemma 6.3.17 again, an compressed representation of Pb can be computed in $n \cdot (\log n \log \Lambda)^{O(1)}$ time. Follows from Fact 6.3.16, $\|Pb\|_1$ can be computed in $n \cdot (\log n \log \Lambda)^{O(1)}$ time. Bp_t can be computed in $O(n + m)$ time. By Lemma 6.3.17, an compressed representation $\widehat{I}$ of PBp_t can be computed in $n \cdot (\log n \log \Lambda)^{O(1)}$ time, and we have $|\widehat{I}| \leq n \cdot (\log n \log \Lambda)^{O(1)}$. By Fact 6.3.16, $\|PBp_t\|_1$ can be computed in $n \cdot (\log n \log \Lambda)^{O(1)}$ time. By

Fact 6.3.16 again, an compressed representation I' of y_t can also be computed in $n\cdot(\log n\log\Lambda)^{O(1)}$ time, and we have $|I'| \leq n\cdot(\log n\log\Lambda)^{O(1)}$. By Lemma 6.3.18, $y_t^\top P$ can be computed in $n\cdot(\log n\log\Lambda)^{O(1)}$ time. To compute $(y_t^\top P)B_i$, we need to compute $(y_t^\top P)A_i$ and $(y_t^\top P)b$. Thus, the running time to compute all $\phi_t^+(i),\phi_t^-(i)$ is $n\cdot(\log n\log\Lambda)^{O(1)}+O(m+n)$. Thus, one iteration of Algorithm 38 takes $m\cdot(\log n\log\Lambda)^{O(1)}$ time. By Lemma 6.3.8, Algorithm 38 takes $\frac{1}{\epsilon^2}\cdot(\log n\log\Lambda)^{O(1)}$ iterations. To construct a $(1+\epsilon,\epsilon/\kappa)$-solver, we need to call Algorithm 38 $\log(\epsilon^{-1}\log\kappa)$ times. Thus, to find an $(1+\epsilon,\epsilon/\kappa)$-solution, the running time is $\frac{m}{\epsilon^2}\cdot(\log n\log\Lambda)^{O(1)}\cdot\log(1/\epsilon)$. Since $\epsilon\geq 1/\Lambda$, the running time is $\frac{m}{\epsilon^2}\cdot(\log n\log\Lambda)^{O(1)}$. Together with Corollary 6.3.5, Corollary 6.3.6 and Lemma 6.3.7, we complete the proof. □

6.4 Implementation in parallel setting

In this section, we will have a detailed discussion of how to implement our algorithms in PRAM model. For convenience, we will describe our algorithms in the PRIORITY CRCW PRAM [73]. In this model, if multiple processors write to the same memory cell, the cell will take the minimum written value. According to [72, 73], algorithms in the PRIORITY CRCW PRAM model can be easily simulated in other PRAM models (including the weakest EREW PRAM model) with at most polylogarithmic factors blow-up in the depth.

6.4.1 Parallel subemulator construction

Theorem 6.4.1 (Parallel construction of subemulator)**.** *Given a connected n-vertex m-edge undirected weighted graph $G=(V,E,w)$ and a parameter $b\in[n]$, there is a PRAM algorithm (Algorithm 33) which outputs an undirected graph $H=(V',E',w')$ and $q:V\to V'$ such that H is a strong $(8,b,22)$-subemulator of G, and q is a corresponding leader mapping (Definition 6.2.1). Furthermore, $\mathbf{E}[|V'|]\leq\min(75\log(n)/b,3/4)n, |E'|\leq nb+m$. The depth of the algorithm is $\log^{O(1)}n$ and the work is $\widetilde{O}(nb^2+m)$.*

Proof. The correctness and the size of H is already shown by Theorem 6.2.9. Next, let us analyze the depth and the work of Algorithm 33.

Consider the depth and the work of Algorithm 31 and Algorithm 32. In Algorithm 31, sampling procedure (line 4) can be done in $O(1)$ depth and $O(n)$ work. For line 5 of Algorithm 31, the implementation is described as the following:

1. For each $v \in V$, compute $r_b(v)$, $\text{Ball}_b(v)$ and $\text{dist}(v,u)$ for $u \in \text{Ball}_b(v)$.
2. For $u \in V$, initialize $l(u) \leftarrow \infty$.
3. If v is sampled to be in S by line 4, let $l(v) \leftarrow 0$ and for each edge $\{v,u\} \in E$, mark $l(u) \leftarrow w(u,v)$. If $l(u)$ is marked multiple times, only keep the minimum one.
4. For $v \in V$, if $\forall u \in \text{Ball}_b(v), \text{dist}(v,u) + l(u) > r_b(v)$, mark v to be in V'.

Due to Lemma 3.1.3, $r_b(v), \text{Ball}_b(v)$ and $\text{dist}(v,u)$ for $u \in \text{Ball}_b(v)$ can be computed in $\log^{O(1)} n$ depth and $\widetilde{O}(nb^2 + m)$ work for all $v \in V$. The last three steps of the above procedure only takes $O(1)$ depth and $O(nb + m)$ work. Thus, Algorithm 31 only uses $\log^{O(1)} n$ depth and $\widetilde{O}(nb^2 + m)$ work.

In Algorithm 32, the implementation of line 3 is similar as line 5 of Algorithm 31:

1. For each $v \in V$, compute $r_b(v)$, $\text{Ball}_b(v)$ and $\text{dist}(v,u)$ for $u \in \text{Ball}_b(v)$.
2. For $u \in V$, initialize $l(u) \leftarrow \infty$.
3. If $v \in V'$, let $l(v) \leftarrow 0, q(v) \leftarrow v$ and for each edge $\{v,u\} \in E$, mark $l(u) \leftarrow w(u,v), q(u) \leftarrow v$. If $l(u)$ is marked multiple times, only keep the minimum one and keep $q(u)$ to be the corresponding v which minimizes $l(u)$ (if there is a tie, let $q(u)$ have the smallest label).
4. For $v \in V$, set $q(v) \leftarrow q(u)$ where $u \in \text{Ball}_b(v)$ and $l(u) + \text{dist}(v,u)$ is minimized. Set $\text{dist}(v, q(v)) \leftarrow l(u) + \text{dist}(v,u)$.

By applying Lemma 3.1.3 again, the above steps only take $\log^{O(1)} n$ depth and $\widetilde{O}(nb^2 + m)$ work. Notice that $\forall v \in V, u \in \text{Ball}_b(v), \text{dist}(v,u)$ is computed, and $\forall v \in V, \text{dist}(v, q(v))$ is also computed. Line 5 of Algorithm 32 has $O(1)$ depth and $O(m)$ work. Line 6 of Algorithm 32 has $O(1)$ depth and

$O(nb)$ work. Line 8 of Algorithm 32 has $O(1)$ depth and $O(m)$ work. Line 9 of Algorithm 32 has $O(1)$ depth and $O(nb)$ work. Overall, Algorithm 32 takes $\log^{O(1)}$ depth and $\widetilde{O}(nb^2 + m)$ work. □

6.4.2 Parallel construction of low hop emulator

Our emulator construction depends on a subroutine PREPROC(G, k) (Algorithm 34). In the following lemma, we analyze the depth and the work of PREPROC(G, k).

Lemma 6.4.2 (Depth and work of PREPROC(G, k))**.** *Given a connected n-vertex m-edge undirected weighted graph $G = (V, E, w)$ with $w : E \to \mathbb{Z}_{\geq 0}$ and a parameter $k \in [0.5, 0.5 \log n]$,* PREPROC*(G, k) (Algorithm 34) has $\log^{O(1)}(n)$ depth and $\widetilde{O}(m + n^{1+1/k})$ expected work.*

Proof. Let t be the value at the end of PREPROC(G, k). According to Theorem 6.4.1, for $i \in \{0, 1, \cdots, t-1\}$, SUBEMULATOR$(H_i, b_i)$ in line 6 of Algorithm 34 takes $\log^{O(1)} n_i$ depth and $\widetilde{O}(n_i b_i^2 + m_i)$ work. Since $n_i \leq n$, the depth is at most $\log^{O(1)} n$. According to Lemma 6.2.10, $\mathbf{E}[n_i b_i^2 + m_i] \leq \max(n^{1+1/k}, n \cdot (75 \log n)^4) + m + 2 \cdot \max(n^{1+1/(2k)}, n \cdot (75 \log n)^2) = \widetilde{O}(m + n^{1+1/k})$. Notice that all the information needed in line 7 of Algorithm 34 can be obtained during SUBEMULATOR(H_i, b_i) (see Lemma 3.1.3 and the proof of Theorem 6.4.1). Since t is at most $O(\log k)$, the overall depth of PREPROC(G, k) is at most $\log^{O(1)} n$ and the overall expected work is at most $\widetilde{O}(m + n^{1+1/k})$. □

Theorem 6.4.3 (Parallel construction of low hop emulator)**.** *Given a connected n-vertex m-edge undirected weighted graph $G = (V, E, w)$ with $w : E \to \mathbb{Z}_{\geq 0}$ and a parameter $k \in [0.5, 0.5 \log n]$, there is a PRAM algorithm (Algorithm 36) which outputs an undirected weighted graph $G' = (V, E', w')$ with $\mathbf{E}[|E'|] \leq O(n^{1+1/(2k)} + n \log^2 n), w' : E' \to \mathbb{Z}_{\geq 0}$ and hop diameter at most $16\lceil \log(k) + 1 \rceil$ such that $\forall u, v \in V$,*

$$\mathrm{dist}_G(u, v) \leq \mathrm{dist}_{G'}(u, v) \leq 27^{4\lceil \log(k)+1 \rceil} \cdot \mathrm{dist}_G(u, v).$$

The depth of the algorithm is at most $\log^{O(1)} n$ and the expected work is at most $\widetilde{O}(m + n^{1+1/k})$.

Proof. The correctness and the size of E' is already analyzed by Theorem 6.2.18. Let us consider the depth and the work of implementing Algorithm 36.

According to Lemma 6.4.2, line 3 of Algorithm 36 has depth $\log^{O(1)} n$ and expected work $\widetilde{O}(m + n^{1+1/k})$. The information needed in line 8 and line 9 of Algorithm 36 can be obtained during PREPROC(G, k) in line 3 of Algorithm 36 (see Algorithm 34, Lemma 3.1.3 and the proof of Theorem 6.4.1). Thus, the remaining steps of Algorithm 36 have $O(1)$ depth and work at most $O(|E'|)$. The overall depth of Algorithm 36 is at most $\log^{O(1)} n$. Since $\mathbf{E}[|E'|] \leq O(n^{1+1/(2k)} + n \log^2 n)$, the expected work of Algorithm 36 is at most $\widetilde{O}(m + n^{1+1/k})$. □

A byproduct of the parallel implementation of PREPROC(G, k) (Algorithm 34) is a parallel distance oracle.

Theorem 6.4.4 (Parallel distance oracle)**.** *Given a connected n-vertex m-edge undirected weighted graph $G = (V, E, w)$ with $w : E \to \mathbb{Z}_{\geq 0}$ and a parameter $k \in [0.5, 0.5 \log n]$, there is a PRAM algorithm (Algorithm 34) which outputs a data structure with expected size $\widetilde{O}\left(n^{1+1/(2k)}\right)$ in depth $\log^{O(1)}(n)$ and expected work $\widetilde{O}(m + n^{1+1/k})$ such that for any pair of vertices $u, v \in V$, a value d satisfying $\text{dist}_G(u, v) \leq d \leq 26^{4\lceil \log(k)+1 \rceil} \text{dist}_G(u, v)$ can be computed in $O(\log(4k))$ time given the outputted data structure.*

Proof. The correctness and the query time is shown by Lemma 6.2.11. By Lemma 6.4.2, PREPROC(G, k) has depth $\log^{O(1)}(n)$ and $\widetilde{O}(m + n^{1/k})$ expected work. Consider QUERY(u, v) (Algorithm 34). Let t be the value at the end of PREPROC(G, k) (Algorithm 34). For $l \in \{0, 1, \ldots, t\}$ and $u \in V_l$, we only need the information $B_l(u)$, $q_l(u)$ and $\text{dist}_{H_l}(u, v)$ for $v \in B_l(\text{u})$. By Lemma 6.2.10, we have $t \leq 4\lceil \log(k) + 1 \rceil$ and $\mathbf{E}[\sum_{l=0}^{t} \sum_{v \in V_l} |B_l(v)|] \leq t \cdot \max(n^{1+1/k}, n \cdot (75 \log n)^4)/b_0 \leq \widetilde{O}(n^{1+1/(2k)})$. Thus the space to store all required information is at most $\widetilde{O}\left(n^{1+1/(2k)}\right)$. □

6.4.3 Direct applications of parallel low hop emulator

poly$(\log n)$-Approximate single source shortest paths (SSSP). A direct application is to compute a poly$(\log n)$-approximate distance from a given vertex s to every other vertex v. We just

need to compute a low hop emulator, and run $O(\log \log n)$ Bellman-Ford iterations starting from the source vertex s.

Corollary 6.4.5 (Parallel poly($\log n$)-approximate SSSP)**.** *Given a connected n-vertex m-edge undirected weighted graph $G = (V, E, w)$ with $w : E \to Z_{\geq 0}$ and a source vertex $s \in V$, there is a PRAM algorithm which can output an* poly($\log n$)*-approximation to* $\mathrm{dist}_G(s, v)$ *for every $v \in V$. Furthermore, the depth of the algorithm is* $\log^{O(1)}(n)$ *and the expected work is* $\widetilde{O}(m)$*.*

Embedding the graph metric into ℓ_1. The second application of our low hop emulator is an efficient parallel algorithm which can embed the graph metric into ℓ_1 space. In particular, given a connected n-vertex m-edge undirected weighted graph $G = (V, E, w)$, we can in parallel find a mapping $\varphi : V \to \ell_1^d$ for $d = O(\log^2 n)$ such that

$$\forall u, v \in V, \mathrm{dist}_G(u, v) \leq \|\varphi(u) - \varphi(v)\|_1 \leq \log^{O(1)}(n) \cdot \mathrm{dist}_G(u, v).$$

This can be done by applying Bourgain's embedding [55] on our low hop emulator. For completeness, the detailed algorithm is described in the following.

Consider a connected n-vertex m-edge undirected weighted graph $G = (V, E, w)$ with $w : E \to Z_{\geq 0}$. By Theorem 6.4.3, we can use expected $\widetilde{O}(m)$ work and $\log^{O(1)}(n)$ depth to compute a low hop emulator $G' = (V, E', w')$, i.e., $\forall u, v \in V$,

$$\mathrm{dist}_G(u, v) \leq \mathrm{dist}_{G'}(u, v) \leq \log^{O(1)}(n) \cdot \mathrm{dist}_G(u, v),$$

and the hop diameter of G' is $O(\log \log n)$. Obviously, the diameter of G' is at most the diameter of G times $\log^{O(1)}(n)$. Furthermore, $\mathbf{E}[|E'|] \leq \widetilde{O}(n)$. According to our construction of G', since all weights in G are integers, weights in G' are also integers.

It suffices to embed G' into ℓ_1. We apply Bourgain's embedding:

1. $t \leftarrow \Theta(\log n)$.

2. For $i = 1 \to \lceil \log n \rceil, j = 1 \to t$:

   (a) Choose a set $S_{i,j}$ by sampling each $v \in V$ with probability 2^{-i}.

   (b) $\forall v \in V$, set the $((i-1) \cdot t + j)$-th coordinate of $\varphi(v)$ as $\mathrm{dist}_{G'}(S_{i,j}, v)$, i.e.,

$$\varphi(v)_{(i-1)\cdot t+j} \leftarrow \mathrm{dist}_{G'}(S_{i,j}, v).$$

.

It is easy to see that for every $v \in V$ the coordinates of $\varphi(v)$ are non-negative integers and $\|\varphi(v)\|_\infty \leq$ the diameter of $G' \leq$ (the diameter of G) $\cdot \log^{O(1)}(n)$. The dimension of $\varphi(v)$ is $t \cdot \lceil \log n \rceil \leq O(\log^2 n)$. By Bourgain's theorem [55], with probability at least 0.999, $\forall u, v \in V$,

$$\mathrm{dist}_{G'}(u, v) \leq \|\varphi(u) - \varphi(v)\|_1 \leq \log^{O(1)}(n) \cdot \mathrm{dist}_{G'}(u, v),$$

which implies that $\forall u, v \in V$,

$$\mathrm{dist}_{G}(u, v) \leq \|\varphi(u) - \varphi(v)\|_1 \leq \log^{O(1)}(n) \cdot \mathrm{dist}_{G}(u, v).$$

Now consider the depth and the work of the above procedure. Step 2b can be implemented by Bellman-Ford algorithm. In particular, we add a super node which connects to every vertex in $S_{i,j}$ with weight zero. By h Bellman-Ford iterations, we can compute $\mathrm{dist}^{(h)}_{G'}(S_{i,j}, v)$ for every $v \in V$. Since the hop diameter of G' is $O(\log \log n)$, we only need $O(\log \log n)$ Bellman-Ford iterations in step 2b. Thus, the depth of the above procedure is at most $\log^{O(1)}(n)$, and the work is at most $\widetilde{O}(|E'|)$. Together with the computation of G', the overall depth is $\log^{O(1)} n$, and the total work is at most $\widetilde{O}(m)$.

Corollary 6.4.6 (Parallel embedding into ℓ_1)**.** *Given a connected n-vertex m-edge undirected weighted graph $G = (V, E, w)$ with $w : E \to Z_{\geq 0}$, there is a PRAM algorithm which can output a mapping $\varphi : V \to [\Delta]^d$ for $\Delta =$ (the diameter of G) $\cdot \log^{O(1)}(n), d = O(\log^2 n)$ such that with*

probability at least 0.99*,*

$$\forall u, v \in V, \mathrm{dist}_G(u, v) \leq \|\varphi(u) - \varphi(v)\|_1 \leq \log^{O(1)}(n) \cdot \mathrm{dist}_G(u, v).$$

Furthermore, the depth of the algorithm is $\log^{O(1)}(n)$ *and the expected work is* $\widetilde{O}(m)$*.*

Low diameter decomposition. Another application is low diameter decomposition. This can be done by applying algorithm of [58] on our low hop emulator. The detailed algorithm is described in the following.

By Theorem 6.4.3, we can use expected $\widetilde{O}(m)$ work and $\log^{O(1)}(n)$ depth to compute a low hop emulator $G' = (V, E', w')$, i.e., $\forall u, v \in V$,

$$\mathrm{dist}_G(u, v) \leq \mathrm{dist}_{G'}(u, v) \leq \log^{O(1)}(n) \cdot \mathrm{dist}_G(u, v),$$

and the hop diameter of G' is $O(\log \log n)$. Furthermore, $\mathbf{E}[|E'|] \leq \widetilde{O}(n)$.

It suffices to run low diameter decomposition [58] on G':

1. For $v \in V$, draw δ_v independently from the exponential distribution with CDF $1 - e^{-\beta x}$.
2. Compute the subset C_u by assigning each v to u which minimizes $\mathrm{dist}_{G'}(v, u) - \delta_u$.
3. Remove empty subsets C_u and return the remaining subsets $\{C_u\}$.

By [58], V will be partitioned into clusters such that

- for any two vertices u, v from the same cluster, $\mathrm{dist}_{G'}(u, v) \leq O(\beta^{-1} \log n)$,
- for any two vertices u, v, the probability that u, v are not in the same cluster is at most $O(\beta \cdot \mathrm{dist}_{G'}(u, v))$.

Thus, it implies that the partition is also good for G:

- for any two vertices u, v from the same cluster, $\mathrm{dist}_G(u, v) \leq \beta^{-1} \log^{O(1)}(n)$,

- for any two vertices u, v, the probability that u, v are not in the same cluster is at most $\beta \cdot \text{dist}_G(u, v) \cdot \log^{O(1)}(n)$.

To implement the second step of the algorithm, we can add a super node which connects to every vertex v with weight $\max_{u \in V} \delta_u - \delta_v$. Then we can use Bellman-Ford to compute the single source shortest path from the super node. Since the hop diameter of G' is at most $O(\log \log n)$, the number of Bellman-Ford iterations is at most $O(\log \log n)$. Thus, the overall work is at most $\widetilde{O}(m)$ and the depth is at most $\log^{O(1)}(n)$.

Corollary 6.4.7 (Low diameter decomposition)**.** *Given a connected n-vertex m-edge undirected weighted graph $G = (V, E, w)$ with $w : E \to Z_{\geq 0}$ and a parameter $\beta \in (0, 1]$, there is a PRAM algorithm which can partition V into subsets $C_1, C_2, \cdots, C_k$ such that*

1. $\forall i \in [k],\ \forall u, v \in C_i,\ \text{dist}_G(u, v) \leq \frac{\log^{O(1)} n}{\beta}$,
2. $\forall u, v \in V,\ \Pr[u, v \text{ are not in the same subset}] \leq \beta \cdot \text{dist}_G(u, v) \cdot \log^{O(1)} n$.

Furthermore, the depth of the algorithm is $\log^{O(1)}(n)$ and the expected work is $\widetilde{O}(m)$.

Metric tree embedding. By applying the parallel FRT embedding (Theorem 7.9 of [57]) on our low hop emulator directly, we can obtain a more work-efficient parallel metric tree embedding algorithm.

Corollary 6.4.8 (Metric tree embedding)**.** *Given a connected n-vertex m-edge undirected weighted graph $G = (V, E, w)$ with $w : E \to \mathbb{Z}_{\geq 0}$, there is a PRAM algorithm which can output a tree $T = (V', E', w')$ where $V' \supseteq V$ such that $\forall u, v \in V$,*

1. $\text{dist}_G(u, v) \leq \text{dist}_T(u, v)$,
2. $\mathbf{E}[\text{dist}_T(u, v)] \leq \log^{O(1)}(n) \cdot \text{dist}_G(u, v)$.

The depth of the algorithm is $\log^{O(1)}(n)$ and the expected work is $\widetilde{O}(m \cdot \log(\text{the diameter of } G))$.

6.4.4 Parallel uncapacitated minimum cost flow

Fact 6.4.9 (Parallel $(n,0)$-solver)**.** *Given a connected n-vertex m-edge undirected weighted graph G and a demand vector $b \in \mathbb{R}^n$,* MSTROUTING(G,b) *(Algorithm 37) can be implemented in PRAM with depth $\log^{O(1)}(n)$ and $\widetilde{O}(m)$ work.*

Proof. We can use Boruvka's algorithm to compute a minimum spanning tree T of G, and it can be implemented in PRAM with $\log^{O(1)}(n)$ depth and $\widetilde{O}(m)$ work (see e.g., [98]). The Euler Tour of T can be computed in $\log^{O(1)}(n)$ depth and $\widetilde{O}(n)$ work [21]. A subtree of T should appear in a consecutive subsequence of the Euler Tour. Thus, the sum of weights in a subtree can be computed as a sum of weights of a subsequence of the Euler Tour. We can use $\log^{O(1)}(n)$ depth and $\widetilde{O}(n)$ work to preprocess a prefix sum over the Euler Tour and hence the line 5 of Algorithm 37 can be computed in $O(1)$ depth and $\widetilde{O}(n)$ work. □

As discussed in Section 6.3, we need to find a good preconditioner.

Lemma 6.4.10 (Work and depth of parallel preconditioner construction)**.** *Given a mapping $\varphi : V \to [\Delta]^d$ for some $\Delta, d \in \mathbb{Z}_{\geq 0}$,* IMPLICITP$(\varphi)$ *(Algorithm 39) can be implemented in PRAM with $(d\log(n\Delta))^{O(1)}$ depth and $n \cdot (d\log(n\Delta))^{O(1)}$ work.*

Proof. Let $L = 1+\log\Delta$. Consider line 4 of Algorithm 39. We can simultaneously handle each pair $(l,v) \in \{0,1,\cdots,L-1\}\times V$. As discussed in the proof of Lemma 6.3.12, $\forall l \in \{0,1,\cdots,L-1\}, v \in V$, we know that $\varphi(v)+\mathbf{1}_d, \varphi(v)+2\cdot\mathbf{1}_d, \cdots, \varphi(v)+2^i\cdot\mathbf{1}_d$ can be in at most $d+1$ different cells in G_i. Notice that each cell can be denoted by one of its corner point and the side length. Thus, the depth of line 4 of Algorithm 39 is $O(d^2)$ and the work is $O(nLd^2)$. For the outer loop and the inner loop started from line 5 and line 7 of Algorithm 39, we can handle each (v,l) simultaneously. Again, there are at most $d+1$ cells in line 8 will be considered, and each cell can be indicated by its side length and one corner point which has d coordinates. In addition, for one cell C, τ_1 and τ_2 can be computed in $O(d)$ time. Thus, line 8 has depth $O(d^2)$ and work $O(nLd^2)$. To implement line 9 of Algorithm 39, for each $l \in \{0,1,\cdots,L-1\}$, we need to index all cells in C_l before the loop started

from line 5. This can be done by sorting. Since $|C_l| \leq (d+1) \cdot n$ and each cell is represented by size at most $O(d)$. The sorting has depth at most $\log^{O(1)}(nd)$ and total work $nL \cdot (d \log n)^{O(1)}$. Notice that we can compute $\sum_{j=0}^{l-1} 2^j |C_j|$ for every $l \in \{0, 1, \cdots, L-1\}$ using depth $O(L)$ and work $O(L)$. After preprocessing steps, the depth of line 9 is $O(d)$ and the work is $O(nLd)$. To conclude, the depth of Algorithm 39 is $(d \log(n\Delta))^{O(1)}$ and the total work is $n \cdot (d \log(n\Delta))^{O(1)}$. □

Lemma 6.4.11 (Work and depth of parallel compressed matrix-vector multiplication)**.** *Given a compressed representation (see Definition 6.3.14) $I = (I_1, I_2, \cdots, I_n)$ of a matrix $P \in \mathbb{R}^{r \times n}$ with $\forall i \in [n], |I_i| \leq s$, and a vector $g \in \mathbb{R}^n$,* MatrixVec(I, g) *(Algorithm 40) can be implemented in PRAM with depth* $\log^{O(1)}(s \cdot \mathrm{nnz}(g))$ *and work* $\widetilde{O}(s \cdot \mathrm{nnz}(g))$.

Proof. For the loop started from line 4 of Algorithm 40, the set S can be created in depth $O(1)$, and the work is $O(|S|)$. The sorting in line 7 has depth $\log^{O(1)} |S|$ and work $|S| \cdot \log^{O(1)} |S|$. For line 8, we need to preprocess a prefix sum $\sum_{t:q_t<q_j} z_t$ for $j \in \{2, 3, \cdots, k\}$. This can be done in depth $\log^{O(1)} |S|$ and work $|S| \log^{O(1)} |S|$. Once we preprocess the prefix sum, we can implement line 8 in $O(1)$ depth and $O(|S|)$ work. Since $|S| = O(s \cdot \mathrm{nnz}(g))$, the overall depth of Algorithm 40 is $\log^{O(1)}(s \cdot \mathrm{nnz}(g))$ and the work is $\widetilde{O}(s \cdot \mathrm{nnz}(g))$. □

Lemma 6.4.12 (Work and depth of parallel compressed vector-matrix multiplication)**.** *Given a compressed representation I of a vector $y \in \mathbb{R}^r$ with $|I| \leq s$ and a compressed representation $I' = (I_1, I_2, \cdots, I_n)$ of a matrix $P \in \mathbb{R}^{r \times n}$ with $\forall i \in [n], |I_i| \leq s'$,* VectorMat$(I, I')$ *(Algorithm 41) can be implemented in PRAM with depth* $\log^{O(1)}(ss')$ *and work* $\widetilde{O}(s + ns')$.

Proof. By sorting, we can implement line 4 and line 5 of Algorithm 41 in $\log^{O(1)}(s)$ depth and $s \log^{O(1)} s$ depth. Line 6 computes the prefix sum p_j for $j \in [s]$. Thus, it needs $\log^{O(1)}(s)$ depth and $s \cdot \log^{O(1)}(s)$ work. For the loop started from line 7, we can handle each $i \in [n]$ and $([a, b], c) \in I_i$ simultaneously. The binary search takes $\log^{O(1)} s$ depth. Line 10 and line 11 compute g_i which is the sum of at most $|I_i| \leq s'$ values. Thus, Line 10 and line 11 have depth $\log^{O(1)}(s')$. To conclude, the overall depth of Algorithm 41 is $\log^{O(1)}(ss')$ and the work is $\widetilde{O}(s + ns')$. □

Theorem 6.4.13 (Parallel uncapacitated minimum cost flow)**.** *Given an $\epsilon \in (0, 0.5)$, a connected n-vertex m-edge undirected weighted graph $G = (V, E, w)$ with $w : E \to \mathbb{Z}_{\geq 0}$, and a demand vector $b \in \mathbb{R}^n$ with $\mathbf{1}_n^\top b = 0$, there is a PRAM algorithm which can output an $(1+\epsilon)$-approximate solution to the uncapacitated minimum cost flow problem with probability at least 0.99. Furthermore, the depth is at most $\epsilon^{-2} \log^{O(1)}(n\Lambda)$ and the expected work is at most $\epsilon^{-2} m \cdot \log^{O(1)}(n\Lambda)$, where $\Lambda = \sum_{e \in E} w(e)$.*

Proof. Let $A \in \mathbb{R}^{n \times m}$ be the vertex-edge incidence matrix of G, and let $W \in \mathbb{R}^{m \times m}$ be the weight matrix. By Corollary 6.4.6, with probability at least 0.99, we can compute a mapping $\varphi : V \to [\Delta]^d$ with $\Delta \leq \Lambda \cdot \log^{O(1)} n, d \leq O(\log^2 n)$ in $\log^{O(1)}(n)$ depth and $\widetilde{O}(m)$ expected work such that

$$\forall u, v \in V, \mathrm{dist}_G(u, v) \leq \|\varphi(u) - \varphi(v)\|_1 \leq \log^{O(1)}(n) \cdot \mathrm{dist}_G(u, v).$$

By Lemma 6.3.15, we can compute a compressed representation $I = \{I_1, I_2, \cdots, I_n\}$ of P with $\forall i \in [n], |I_i| \leq O(d \log \Delta) = \log^{O(1)}(n\Lambda)$. Furthermore, $\kappa(PAW^{-1}) \leq \log^{O(1)}(n\Lambda)$. By Lemma 6.4.10, the depth of computing such compressed representation is $\log^{O(1)}(n\Lambda)$ and the work is $n \cdot \log^{O(1)}(n\Lambda)$. Now, we are able to implement Algorithm 38 in parallel. To compute matrix B, we need to compute $\|PAW^{-1}\|_{1\to 1}$ and $\|Pb\|_1$. Notice that $\|PAW^{-1}\|_{1\to 1} = \max_{i \in [m]} \|P(AW^{-1})_i\|_1$ and $(AW^{-1})_i$ only has two non-zero entries. By Lemma 6.3.17 and Lemma 6.4.11, a compressed representation of $P(AW^{-1})_i$ can be computed in $\log^{O(1)}(n\Lambda)$ depth. We can compute $P(AW^{-1})_i$ for all $i \in [m]$ simultaneously. By Lemma 6.4.11, the total work needed is $m \cdot \log^{O(1)}(n\Lambda)$. By Lemma 6.3.17, the size of the compressed representation of $P(AW^{-1})_i$ is at most $\log^{O(1)}(n\Lambda)$. Thus, by Fact 6.3.16, $\|P(AW^{-1})_i\|_1$ can be computed in $\log^{O(1)}(n\Lambda)$ depth. Thus, $\|PAW^{-1}\|_{1\to 1}$ can be computed in $\log^{O(1)}(n\Lambda)$ depth and $m \cdot \log^{O(1)}(n\Lambda)$ work. By Lemma 6.3.17 and Lemma 6.4.11 again, a compressed representation of Pb can be computed in $\log^{O(1)}(n\Lambda)$ depth and $n \cdot \log^{O(1)}(n\Lambda)$ work. By Lemma 6.3.17, the size of the compressed representation of Pb is at most $n \cdot \log^{O(1)}(n\Lambda)$, thus $\|Pb\|_1$ can be computed by the summation of at most $n \cdot \log^{O(1)}(n\Lambda)$ values. Such summation operation has at most $\log^{O(1)}(n\Lambda)$ depth and $n \cdot \log^{O(1)}(n\Lambda)$ work. Algorithm 38 has $O(\epsilon^{-2}\kappa^2 \log n) =$

$\epsilon^{-2}\log^{O(1)}(n\Lambda)$ iterations. In each iteration, we firstly need to compute p_t. This can be done in $\log^{O(1)}(n)$ depth and $m\log^{O(1)}(n)$ work. Notice that AW^{-1} only has $O(m)$ non-zero entries and $b\cdot\mathbf{1}_m^\top p_t = b$. We can compute Bp_t in depth $\log^{O(1)}(n)$ and work $m\cdot\log^{O(1)} n$. By Lemma 6.3.17 and Lemma 6.4.11, a compressed representation $\widehat{I}$ of PBp_t can be computed in depth $\log^{O(1)}(n\Lambda)$ and work $n\cdot\log^{O(1)}(n\Lambda)$. By Lemma 6.3.17, the size of $\widehat{I}$ is at most $n\log^{O(1)}(n\Lambda)$. Similar as computing $\|Pb\|_1$, $\|PBp_t\|_1$ can be computed in depth $\log^{O(1)}(n\Lambda)$ and work $n\log^{O(1)}(n\Lambda)$. Now, we want to compute a compressed representation I' of y_t. To achieve this, we can look at each $([a,b],c)\in\widehat{I}$ simultaneously and put $([a,b],\mathrm{sgn}(c))$ into I'. This step has depth $O(1)$ and work at most $n\log^{O(1)}(n\Lambda)$. It is easy to see $|I'| = |\widehat{I}|$. By Lemma 6.3.18 and Lemma 6.4.12, $y_t^\top P$ can be computed in depth $\log^{O(1)}(n\Lambda)$ and work $n\cdot\log^{O(1)}(n\Lambda)$. To compute $(y_t^\top P)B_i$, we need to compute $(y_t^\top P)A_i$ and $(y_t^\top P)b$. Since A_i has at most 2 non-zero entries, we can simultaneously compute $(y_t^\top P)A_i$ for all $i\in[m]$ and $(y_t^\top P)b$. It has depth $\log^{O(1)}(n\Lambda)$ and $m\log^{O(1)}(n\Lambda)$ work. To construct a $(1+\epsilon,\epsilon/\kappa)$-solver, we need to invoke Algorithm 38 $\log(\epsilon^{-1}\log\kappa)$ times. Thus, to find an $(1+\epsilon,\epsilon/\kappa)$-solution, the total depth is $\epsilon^{-2}\log^{O(1)}(\epsilon^{-1}n\Lambda)$ and the work is $\epsilon^{-2}m\log^{O(1)}(\epsilon^{-1}n\Lambda)$. Since $\epsilon\geq 1/\Lambda$, $\log(\epsilon^{-1})\leq\log\Lambda$. Together with Corollary 6.3.5, Corollary 6.3.6, Lemma 6.3.7 and Fact 6.4.9, we complete the proof. □

6.4.5 Parallel $s-t$ approximate shortest path

Given two vertices s and t, a special case of uncapacitated minimum cost flow is when the demand vector b only has 2 non-zero entries: $b_s = 1$ and $b_t = -1$. In this case, the value of the minimum cost flow is exactly the same as the distance between s and t. As shown previously, since we can compute a $(1+\epsilon)$-approximation to the uncapacitated minimum cost flow, we can compute a $(1+\epsilon)$-approximation to the distance between s and t. However, our flow algorithm can only output a flow from s to t. In this section, we will show how to obtain an $s-t$ path from the $s-t$ flow.

Before we present our algorithm, let us show a good property of the random walk corresponding to the flow. Consider a connected n-vertex m-edge undirected weighted graph $G=(V,E,w)$.

Let $A \in \mathbb{R}^{n\times m}$ be the corresponding vertex-edge incidence matrix (see Section 6.3), and let $W \in \mathbb{R}^{m\times m}$ be the corresponding diagonal weight matrix. Given a valid demand vector $b \in \mathbb{R}^n$, i.e., $\mathbf{1}_n^\top b = 0$, let $f \in \mathbb{R}^m$ be a feasible flow, i.e., $Af = b$. Suppose $\{u, v\} \in E$ is the i-th edge. We denote $f(u, v)$ as the flow from u to v, i.e.,

$$f(u,v) = \begin{cases} f_i & u < v, \\ -f_i & u > v. \end{cases}$$

By the definition of $f(u, v)$, we have $f(u, v) = -f(v, u)$. For $\{u, v\} \notin E$, we denote $f(u, v) = 0$. If $f(u, v)$ is negative, then it means that there is $-f(u, v)$ units of flow from v to u. Suppose the demand vector b further satisfies $b_t = -1$, and $\forall v \neq t, b_v \geq 0$. Notice that since $\mathbf{1}_n^\top b = 0$, we have $\sum_{v\in V\setminus\{t\}} b_v = 1$. We can generate a random walk by the following way:

1. Set $i \leftarrow 0$ and set u_0 to be $v \in V \setminus \{t\}$ with probability b_v.
2. Set u_{i+1} to be $v \in \{v' \in V \mid f(u_i, v') > 0\}$ with probability $\frac{f(u_i,v)}{\sum_{v':f(u_i,v')>0} f(u_i,v')}$.
3. If $u_{i+1} \neq t$, set $i \leftarrow i + 1$, and repeat step 2. Otherwise, output the path $p = (u_0, u_1, \cdots, u_{i+1})$.

We say the path p is a random walk corresponding to the flow f. If the flow f contains no cycle, then the expected length of p is exactly the same as the cost of f (see e.g., [43]). The following lemma shows that if f contains cycles then the expected length of p is still the cost of f.

Lemma 6.4.14 (Expected length of a random walk)**.** *Consider a connected n-vertex m-edge undirected weighted graph $G = (V, E, w)$ with $w : E \to \mathbb{Z}_{\geq 0}$. Let $A \in \mathbb{R}^{n\times m}$ be the corresponding vertex-edge incidence matrix, and let $W \in \mathbb{R}^{m\times m}$ be the corresponding diagonal weight matrix. Given a vertex $t \in V$ and a demand vector $b \in \mathbb{R}^n$ satisfying $b_t = -1$ and $\forall v \neq t, b_v \geq 0$, let $f \in \mathbb{R}^m$ be a feasible flow for b, i.e., $Af = b$. The expected length of a random walk corresponding to the flow f is $\|Wf\|_1$.*

Proof. For $u \in V$, let $d(u)$ denote the expected length of a random walk starting from the vertex u.

Notice that $d(t) = 0$. For each vertex $u \in V$, we have the following equation:

$$d(u) = \sum_{v:f(u,v)>0} \frac{f(u,v)}{\sum_{v':f(u,v')>0} f(u,v')} \cdot (d(v) + w(u,v)).$$

By reordering the terms, we have:

$$\left(\sum_{v':f(u,v')>0} f(u,v') \cdot d(u)\right) - \left(\sum_{v:f(u,v)>0} f(u,v) \cdot d(v)\right) = \sum_{v:f(u,v)>0} f(u,v) \cdot w(u,v).$$

By summation over all vertices $u \in V$, we have:

$$\begin{aligned}
&\sum_{u\in V}\left(\left(\sum_{v':f(u,v')>0} f(u,v') \cdot d(u)\right) - \left(\sum_{v:f(u,v)>0} f(u,v) \cdot d(v)\right)\right) = \sum_{u\in V}\sum_{v:f(u,v)>0} f(u,v) \cdot w(u,v) \\
\Rightarrow &\sum_{u\in V} d(u) \cdot \left(\left(\sum_{v':f(u,v')>0} f(u,v')\right) - \left(\sum_{v:f(v,u)>0} f(v,u)\right)\right) = \|Wf\|_1 \\
\Rightarrow &\sum_{u\in V} d(u) \cdot b_u = \|Wf\|_1.
\end{aligned}$$

Since $d(t) = 0$, the expected length of a random walk corresponding to f is $\sum_{u\in V\setminus\{t\}} d(u) \cdot b_u = \sum_{u\in V} d(u) \cdot b_u = \|Wf\|_1$. □

According to the above lemma, if f is a $(1+\epsilon)$-approximation to the optimal $s-t$ flow, then an $s-t$ random walk corresponding to f is a $(1+\epsilon)$-approximate shortest path. However, simulating a random walk in parallel may not be easy. Instead, we will show how to in parallel find a path of which expected length is at most the expected length of a random walk (Algorithm 42).

Lemma 6.4.15 (Work and depth of parallel $s-t$ approximate shortest path)**.** *Given an $\epsilon \in (0, 0.5)$, a connected n-vertex m-edge undirected graph $G = (V, E, w)$ with $w : E \to \mathbb{Z}_{\geq 0}$, and two vertices $s, t \in V$,* FindPath(G, s, t, ϵ) *(Algorithm 42) can be implemented in PRAM with $\epsilon^{-2}\,\mathrm{poly}(\log(n\Lambda))$ depth and expected $\epsilon^{-2}m\,\mathrm{poly}(\log(n\Lambda))$ work, where $\Lambda = \max_{e\in E} w(e)$.*

Proof. Line 5 can be done in $\epsilon^{-2}\,\mathrm{poly}(\log(n\Lambda))$ depth using expected $\epsilon^{-2}m\,\mathrm{poly}(\log(n\Lambda))$ work by

Algorithm 42 Finding an $s - t$ Path

1: **procedure** FINDPATH($G = (V, E, w), s, t \in V, \epsilon \in (0, 0.5)$)
2: Output: $p = (u_0, u_1, u_2, \cdots, u_h)$
3: If $s = t$, return $p = (s)$.
4: $n \leftarrow |V|, m \leftarrow |E|$. Initialize a demand vector $b \in \mathbb{R}^n$: $b_s \leftarrow 1, b_t \leftarrow -1, \forall v \neq s, t, b_v \leftarrow 0$.
5: Compute a $(1+\epsilon)$-approximate uncapacitated minimum cost flow f satisfying b. ▷ Theorem 6.4.13.
6: For each vertex $u \in V \setminus \{t\}$, set the pointer $\mathrm{par}(u) \leftarrow v \in \{v' \in V \mid f(u, v') > 0\}$ with probability

$$\frac{f(u, v)}{\sum_{v': f(u,v')>0} f(u, v')}.$$

7: Let $G' = (V, E')$, where $E' = \{\{u, \mathrm{par}(u)\} \mid u \in V \setminus \{t\}\}$.
8: Compute a spanning forest of G'. For $u \in V$, if u is in the same connected component as t, set $\mathrm{root}(u) \leftarrow t$; otherwise set $\mathrm{root}(u) \leftarrow v$ where v is in the same connected component as u and the edge $\{v, \mathrm{par}(v)\}$ does not appear in the spanning forest.
9: For $u \in V$, compute $l(u) \leftarrow w(u, \mathrm{par}(u)) + w(\mathrm{par}(u), \mathrm{par}(\mathrm{par}(u))) + \cdots + w(\mathrm{par}(\cdots \mathrm{par}(u)), \mathrm{root}(u))$.
10: Set $V'' \leftarrow \{v \in V \mid \mathrm{root}(v) = v\}, E'' \leftarrow \{\{\mathrm{root}(u), \mathrm{root}(v)\} \mid \{u, v\} \in E\}$.
11: For each $e'' = \{u'', v''\} \in E''$, set

$$\mathrm{map}(e'') \leftarrow \arg \min_{\substack{\{u,v\} \in E: \\ \mathrm{root}(u)=u'', \mathrm{root}(v)=v''}} l(u) + w(u, v) + l(v).$$

12: For each $e'' \in E''$, set $w''(e'') \leftarrow l(u) + w(u, v) + l(v)$, where $\{u, v\} = \mathrm{map}(e'')$.
13: $p'' = (u''_0, u''_1, \cdots, u''_{h''}) \leftarrow$FINDPATH$(G'' = (V'', E'', w''), \mathrm{root}(s), \mathrm{root}(t), \epsilon)$. ▷ Recursion.
14: Create p by replacing u''_0 with

$$(s, \mathrm{par}(s), \mathrm{par}(\mathrm{par}(s)), \cdots, \mathrm{root}(s)),$$

and replacing each edge $\{u''_{i-1}, u''_i\}$ of p'' with a path

$$(\mathrm{root}(x_i), \mathrm{par}(\cdots \mathrm{par}(x_i)), \cdots, \mathrm{par}(x_i), x_i, y_i, \mathrm{par}(y_i), \mathrm{par}(\mathrm{par}(y_i)), \cdots, \mathrm{root}(y_i)),$$

where $\{x_i, y_i\} = \mathrm{map}(\{u''_{i-1}, u''_i\})$.
15: Shortcut all cycles of p and return p.
16: **end procedure**

Theorem 6.4.13. We can repeat line 5 $\Theta(\log n)$ times to boost the success probability to $1 - n^{-10}$. It only increases the work by a $O(\log n)$ factor. Line 6 can be done in $\mathrm{poly}(\log n)$ depth using $m\,\mathrm{poly}(\log n)$ work. In line 8, computing connected components and a spanning forest can be done in $\mathrm{poly}(\log n)$ depth using $m\,\mathrm{poly}(\log n)$ work [13]. In line 9, we can use doubling algorithm to compute $l(u)$ for all $u \in V$ simultaneously in $\mathrm{poly}(\log n)$ depth using $n\,\mathrm{poly}(\log n)$ work. Suppose the number of hops of p before shortcutting cycles is h. Then the path p in line 14 can be obtained

in poly(log(nh)) depth using $(n+h)\cdot$ poly(log(nh)) work by a doubling algorithm. Notice that the path p'' obtained by line 13 has no cycle and thus each edge $e \in E$ can appear in p obtained by line 14 at most twice. Hence, line 14 can be implemented in poly(log n) depth using m poly(log n) work. In line 15, we use the following way to shortcut cycles of p. We construct a graph which consists of all edges in p. We find a spanning tree of the obtained graph and output the desired path on the tree. The total number of edges in the path before shortcutting is at most $O(m)$ as as discussed. We use [13] again to find a spanning tree. It takes poly(log(n)) depth and $m\cdot$ poly(log n) total work. And we use doubling algorithm again to find the path, it takes poly(log(n)) depth and $n\cdot$ poly(log n) total work.

Next, consider the number of recursions (line 13) needed. Except vertex t, every vertex u has an edge $\{u, \text{par}(u)\} \in E'$. Thus, each connected component without t in G' has size at most 2. Notice that $|V''|$ is exactly the same as the number of connected components in G'. Thus, $|V''| \le \lceil n/2 \rceil$. It implies that the number of recursions is at most $\lceil \log n \rceil$. Thus, the overall depth is at most ϵ^{-2} poly(log($n\Lambda$)) and the overall expected work is at most $\epsilon^{-2} m$ poly(log($n\Lambda$)). □

Lemma 6.4.16 (Correctness of parallel approximate $s-t$ shortest path)**.** *Given an $\epsilon \in (0, 0.5)$, a connected n-vertex m-edge undirected graph $G = (V, E, w)$ with $w : E \to \mathbb{Z}_{\ge 0}$, and two vertices $s, t \in V$, let p be the output of* FindPath(G, s, t, ϵ) *(Algorithm 42).* $\mathbf{E}[w(p)] \le (1+2\epsilon)^{\lceil \log n \rceil} \cdot \text{dist}_G(s,t)$.

Proof. Our proof is by induction on the number of vertices of G. When G only has one vertex, the statement is obviously true. Now suppose the statement is true for any graph with less than n vertices and consider G with n vertices.

For $u \in V$, let $l(u)$ be the same as that in line 9. Let p be the output of FindPath(G, s, t, ϵ). Let $p'' = (u''_0, u''_1, \cdots, u''_{h''})$ be the output of line 13. By line 13, $u''_0 = \text{root}(s), u''_{h''} = \text{root}(t)$. By line 8, we further have $u''_{h''} = \text{root}(t) = t$.

Claim 6.4.17. $w(p) \le w''(p'') + l(s)$.

Proof. Notice that

$$\begin{aligned}
w(p) &\le w((s, \mathrm{par}(s), \mathrm{par}(\mathrm{par}(s)), \cdots, \mathrm{root}(s))) \\
&\quad + \sum_{\substack{x_i, y_i: \\ \{x_i, y_i\} = \mathrm{map}(\{u''_{i-1}, u''_i\})}} w((\mathrm{root}(x_i), \cdots, \mathrm{par}(x_i), x_i, y_i, \mathrm{par}(y_i), \cdots, \mathrm{root}(y_i))) \\
&= l(s) + \sum_{\substack{x_i, y_i: \\ \{x_i, y_i\} = \mathrm{map}(\{u''_{i-1}, u''_i\})}} l(x_i) + w(x_i, y_i) + l(y_i) \\
&= l(s) + \sum_{i=1}^{h''} w''(u''_{i-1}, u''_i) \\
&= l(s) + w''(p''),
\end{aligned}$$

where the first inequality follows from line 15, the first equality follows from line 9, and the second equality follows from line 12. □

Claim 6.4.18. $\mathbf{E}_{\mathrm{par}: V \setminus \{t\} \to V}[l(s) + \mathrm{dist}_{G''}(\mathrm{root}(s), \mathrm{root}(t))] \le (1 + 2\epsilon)\,\mathrm{dist}_G(s, t)$.

Proof. Let f be the flow obtained by line 5 satisfying $\sum_{\{u,v\} \in E} w(u, v) \cdot |f(u, v)| \le (1+\epsilon)\,\mathrm{dist}_G(s, t)$. Our proof is by coupling. By Lemma 6.4.14, we only need to prove that $\mathbf{E}[l(s)+\mathrm{dist}_{G''}(\mathrm{root}(s), \mathrm{root}(t))]$ is almost upper bounded by the expected length of a random walk corresponding to the flow f.

For $u \in V$, let $\mathrm{par}(u)$ be the same as that in Algorithm 42. We conceptually generate a random walk $\widehat{p}$ corresponding to f in the following way:

1. Set $i \leftarrow 0$ and set $\widehat{u}_0 \leftarrow s$.
2. If $\forall j \in \{0, 1, \cdots, i-1\}, \widehat{u}_j \ne \widehat{u}_i$, then set $\widehat{u}_{i+1} \leftarrow \mathrm{par}(\widehat{u}_i)$. Otherwise, set $\widehat{u}_{i+1}$ to be $v \in \{v' \in V \mid f(\widehat{u}_i, v') > 0\}$ with probability $\frac{f(\widehat{u}_i, v)}{\sum_{v': f(\widehat{u}_i, v') > 0} f(\widehat{u}_i, v')}$.
3. If $\widehat{u}_{i+1} \ne t$, set $i \leftarrow i + 1$, and repeat step 2. Otherwise, $\widehat{h} \leftarrow i + 1$, and output the path $\widehat{p} = (\widehat{u}_0, \widehat{u}_1, \cdots, \widehat{u}_{\widehat{h}})$ as the random walk.

It is easy to see that $\widehat{p}$ is a random walk corresponding to the flow f and thus $\mathbf{E}[w(\widehat{p})] = \sum_{\{u,v\} \in E} w(u, v) \cdot$

$|f(u,v)|$ by Lemma 6.4.14. Notice that $\widehat{p}$ may have cycles, we conceptually do the following procedure to shortcut all cycles of $\widehat{p}$ to obtain $\bar{p}$:

1. Set $i \leftarrow 0, j \leftarrow 0$.
2. If $\nexists i' > i, \widehat{u}_{i'} = \widehat{u}_i$, set $\bar{u}_j \leftarrow \widehat{u}_i, j \leftarrow j+1, i \leftarrow i+1$.
3. Otherwise, find the largest $i' \in [\widehat{h}]$ such that $\widehat{u}_{i'} = \widehat{u}_i$. Mark all edges $\{\widehat{u}_i, \widehat{u}_{i+1}\}$, $\{\widehat{u}_{i+1}, \widehat{u}_{i+2}\}$, $\cdots$, $\{\widehat{u}_{i'-1}, \widehat{u}_{i'}\}$ as *redundant*.
4. Repeat above two steps until $i > \widehat{h}$. At the end, set $\bar{h} \leftarrow j-1$ and path $\bar{p} \leftarrow (\bar{u}_0, \bar{u}_1, \cdots, \bar{u}_{\bar{h}})$. For each edge $e \in E$ which appears in $\widehat{p}$ but is not marked as redundant, mark e as *crucial*.

It is easy to see that $\bar{p}$ is a simple s-t path and furthermore $\sum_{e \in E: e \text{ is crucial}} w(e) \leq w(\bar{p}) \leq w(\widehat{p}) - \sum_{e \in E: e \text{ is redundant}} w(e)$. By the above constructions, $\widehat{p}$ has several good properties.

Fact 6.4.19. *If $\widehat{u}_i$ is the first appearance of a vertex v in $\widehat{p}$, then $\widehat{u}_{i+1}$ must be* $\mathrm{par}(v)$.

Fact 6.4.20. *If vertex $v \in V$ appears in $\widehat{p}$ and $v \neq t$, all edges $\{v, \mathrm{par}(v)\}, \{\mathrm{par}(v), \mathrm{par}(\mathrm{par}(v))\}$, $\{\mathrm{par}(\mathrm{par}(v)), \mathrm{par}(\mathrm{par}(\mathrm{par}(v)))\}, \cdots$, should be in $\widehat{p}$.*

Proof. It directly follows from Fact 6.4.19. □

Fact 6.4.21. *If vertex v appears at least twice in $\widehat{p}$, $\{v, \mathrm{par}(v)\}$ is marked as redundant.*

Proof. Suppose $\widehat{u}_i$ is the first appearance of v, then $\widehat{u}_{i+1} = \mathrm{par}(v)$ (Fact 6.4.19). Suppose $\widehat{u}_{i'}$ is the second appearance of v, we have $\widehat{u}_i = \widehat{u}_{i'}$ and $i' > i$. It is easy to verify that $\{\widehat{u}_i, \widehat{u}_{i+1}\}$ is marked as redundant in the procedure of constructing $\bar{p}$ in any case. □

Fact 6.4.22. *If $\{v, \mathrm{par}(v)\}$ is a redundant edge, $\{\mathrm{par}(v), \mathrm{par}(\mathrm{par}(v))\}$ is a redundant edge.*

Proof. If $\mathrm{par}(v)$ appears in $\widehat{p}$ at least twice, $\{\mathrm{par}(v), \mathrm{par}(\mathrm{par}(v))\}$ is a redundant edge due to Fact 6.4.21. Otherwise, suppose $\widehat{u}_i$ is the first appearance of v. $\widehat{u}_{i+1}$ must be $\mathrm{par}(v)$ due to Fact 6.4.19. Since $\mathrm{par}(v)$ only appears once, $\widehat{u}_{i+1}$ is the first appearance of $\mathrm{par}(v)$. By Fact 6.4.19 again, $\widehat{u}_{i+2}$ must be $\mathrm{par}(\mathrm{par}(v))$. Since $\{\widehat{u}_i, \widehat{u}_{i+1}\}$ is a redundant edge and $\widehat{u}_{i+1}$ only appears once, by the procedure of

constructing $\bar{p}$, $\{\widehat{u}_{i+1}, \widehat{u}_{i+2}\}$ must be marked as redundant in the same step when $\{\widehat{u}_i, \widehat{u}_{i+1}\}$ is marked as redundant. Thus, $\{\mathrm{par}(v), \mathrm{par}(\mathrm{par}(v))\}$ is a redundant edge. □

Fact 6.4.23.

$$\mathop{\mathbf{E}}_{\widehat{p}}\left[\sum_{e\in E: e \text{ is redundant}} w(e)\right] \leq \epsilon \cdot \mathrm{dist}_G(s,t).$$

Proof. Notice that

$$\sum_{e\in E: e \text{ is redundant}} w(e) \leq w(\widehat{p}) - w(\bar{p}) \leq w(\widehat{p}) - \mathrm{dist}_G(s,t),$$

where the last inequality follows from that $w(\bar{p}) \geq \mathrm{dist}_G(s,t)$.

On the other hand, we have:

$$\mathbf{E}\left[w(\widehat{p})\right] = \sum_{\{u,v\}\in E} w(u,v)\cdot|f(u,v)| \leq (1+\epsilon)\,\mathrm{dist}_G(s,t),$$

where the first equality follows from Lemma 6.4.14 and the last inequality follows from that f is a $(1+\epsilon)$-approximate uncapacitated minimum cost $s-t$ flow. Thus,

$$\mathbf{E}\left[\sum_{e\in E: e \text{ is redundant}} w(e)\right] \leq \epsilon \cdot \mathrm{dist}_G(s,t).$$

□

Next, we show how to find a $\mathrm{root}(s) - \mathrm{root}(t)$ path in G'' of which cost plus $l(s)$ is at most $\sum_{e\in E: e \text{ is crucial}} w(e) + 2\sum_{e\in E: e \text{ is redundant}} w(e)$. Let us conceptually construct a path $\widetilde{p}$ in G using the following way:

1. Initialize $\widetilde{p} = (s, \mathrm{par}(s), \mathrm{par}(\mathrm{par}(s)), \cdots, \mathrm{root}(s)), i \leftarrow 0$. Let $\widetilde{u}''_0 \leftarrow \mathrm{root}(s)$.

2. Let $\widehat{u}_{k_i}$ be the last vertex in $\widehat{p}$ such that $\widehat{u}_{k_i}$ is in the same connected component as $\widetilde{u}''_i$ in G' (G' is constructed by line 7).

3. If $\widetilde{u}''_i = t$, let $\widetilde{h}'' \leftarrow i$ and finish the procedure.

4. Otherwise, concatenate

$$(\mathrm{root}(\widehat{u}_{k_i}) = \widetilde{u}''_i, \cdots, \mathrm{par}(\mathrm{par}(\widehat{u}_{k_i})), \mathrm{par}(\widehat{u}_{k_i}), \widehat{u}_{k_i}, \widehat{u}_{k_i+1}, \mathrm{par}(\widehat{u}_{k_i+1}), \mathrm{par}(\mathrm{par}(\widehat{u}_{k_i+1})), \cdots, \mathrm{root}(\widehat{u}_{k_i+1}))$$

to $\widetilde{p}$ and set $\widetilde{u}''_{i+1} \leftarrow \mathrm{root}(\widehat{u}_{k_i+1})$.

5. Set $i \leftarrow i + 1$ and go to step 3.

Fact 6.4.24. $\forall i \neq j \in \{0, 1, \cdots, \widetilde{h}''\}, \widetilde{u}''_i \neq \widetilde{u}''_j$.

Proof. This follows from that $\widehat{u}_{k_i}$ is the last vertex in $\widehat{p}$ such that $\widehat{u}_{k_i}$ is in the same connected component as $\widetilde{u}''_i$ in G' and $k_0 < k_1 < k_2 < \cdots < k_{\widehat{h}''}$. □

Fact 6.4.25. *Each edge in $\widetilde{p}$ appears in $\widehat{p}$.*

Proof. Since $\widehat{u}_0 = s$, edges $\{s, \mathrm{par}(s)\}, \{\mathrm{par}(s), \mathrm{par}(\mathrm{par}(s))\}, \{\mathrm{par}(\mathrm{par}(s)), \mathrm{par}(\mathrm{par}(\mathrm{par}(s)))\}, \cdots$ must appear in $\widehat{p}$ by Fact 6.4.20. By the choice of $\widehat{u}_{k_i}$ and $\widehat{u}_{k_i+1}$, edge $\{\widehat{u}_{k_i}, \widehat{u}_{k_i+1}\}$ appears in $\widehat{p}$. By Fact 6.4.20 again, edges $\{\widehat{u}_{k_i}, \mathrm{par}(\widehat{u}_{k_i})\}, \{\mathrm{par}(\widehat{u}_{k_i}), \mathrm{par}(\mathrm{par}(\widehat{u}_{k_i}))\}, \{\mathrm{par}(\mathrm{par}(\widehat{u}_{k_i})), \mathrm{par}(\mathrm{par}(\mathrm{par}(\widehat{u}_{k_i})))\}, \cdots$ and edges $\{\widehat{u}_{k_i+1}, \mathrm{par}(\widehat{u}_{k_i+1})\}, \{\mathrm{par}(\widehat{u}_{k_i+1}), \mathrm{par}(\mathrm{par}(\widehat{u}_{k_i+1}))\}, \{\mathrm{par}(\mathrm{par}(\widehat{u}_{k_i+1})), \mathrm{par}(\mathrm{par}(\mathrm{par}(\widehat{u}_{k_i+1})))\}, \cdots$ also appear in $\widehat{p}$. □

Fact 6.4.26. *Each edge in $\widetilde{p}$ can appear at most twice. In addition, if an edge e in $\widetilde{p}$ appears twice, e is a redundant edge.*

Proof. Suppose $e = \{u, v\}$ appears in $\widetilde{p}$. By Fact 6.4.24, $\mathrm{root}(u), \mathrm{root}(v)$ only appears once in $\widetilde{p}$. If u and v are not in the same connected component in G', then either $u = \widehat{u}_{k_i}, v = \widehat{u}_{k_i+1}$ or $v = \widehat{u}_{k_i}, u = \widehat{u}_{k_i+1}$ for some $i \in \{0, 1, \cdots, \widetilde{h}''\}$. In this case, $\{u, v\}$ only appears once in $\widetilde{p}$. If u and v are in the same connected component, then either $v = \mathrm{par}(u)$ or $u = \mathrm{par}(v)$. Without loss of generality, suppose $v = \mathrm{par}(u)$. If $(u, \mathrm{par}(u))$ appears in $\widetilde{p}$, the subpath $(u, \mathrm{par}(u), \mathrm{par}(\mathrm{par}(u)), \cdots, \mathrm{root}(u))$ appears in $\widetilde{p}$. If $(\mathrm{par}(u), u)$ appears in $\widetilde{p}$, the subpath $(\mathrm{root}(u), \cdots, \mathrm{par}(\mathrm{par}(u)), \mathrm{par}(u), u)$ appears in $\widetilde{p}$. As mentioned previously, $\mathrm{root}(u)$ only appears once in $\widetilde{p}$. Thus, $(u, \mathrm{par}(u))$ can appear at most

once in $\widetilde{p}$ and $(\mathrm{par}(u), u)$ can appear at most once in $\widetilde{p}$ which means that $\{u, \mathrm{par}(u)\}$ can appear at most twice in $\widetilde{p}$.

Suppose $\{u, \mathrm{par}(u)\}$ appears twice in $\widetilde{p}$, then by the construction of $\widetilde{p}$, the subpath

$$(u, \mathrm{par}(u), \mathrm{par}(\mathrm{par}(u)), \cdots, \mathrm{root}(u), \cdots, \mathrm{par}(\mathrm{par}(u)), \mathrm{par}(u), u) \tag{6.15}$$

must appear in $\widetilde{p}$. More precisely, the subpath (6.15) should appear in the subpath

$$(\widehat{u}_{k_i+1}, \mathrm{par}(\widehat{u}_{k_i+1}), \cdots, u, \mathrm{par}(u), \cdots, \mathrm{root}(u), \cdots, \mathrm{par}(u), u, \cdots, \mathrm{par}(\widehat{u}_{k_{i+1}}), \widehat{u}_{k_{i+1}})$$

for some $i \in \{0, 1, 2, \cdots, \widetilde{h}'' - 1\}$. If $\widehat{u}_{k_i+1} = \widehat{u}_{k_{i+1}}$, it means that $\widehat{u}_{k_i+1}$ appears twice in $\widehat{p}$. In this case, by Fact 6.4.21, $\{\widehat{u}_{k_i+1}, \mathrm{par}(\widehat{u}_{k_i+1})\}$ is a redundant edge. By Fact 6.4.22,

$$\{\widehat{u}_{k_i+1}, \mathrm{par}(\widehat{u}_{k_i+1})\}, \{\mathrm{par}(\widehat{u}_{k_i+1}), \mathrm{par}(\mathrm{par}(\widehat{u}_{k_i+1}))\}, \{\mathrm{par}(\mathrm{par}(\widehat{u}_{k_i+1})), \mathrm{par}(\mathrm{par}(\mathrm{par}(\widehat{u}_{k_i+1})))\}, \cdots$$

are redundant edges. Thus, $\{u, \mathrm{par}(u)\}$ is a redundant edge. In the case when $\widehat{u}_{k_i+1} \neq \widehat{u}_{k_{i+1}}$, we can find two vertices $x \neq y$ such that $x = \mathrm{par}(\mathrm{par}(\cdots \mathrm{par}(\widehat{u}_{k_i+1})))$, $y = \mathrm{par}(\mathrm{par}(\cdots \mathrm{par}(\widehat{u}_{k_{i+1}})))$ and $\mathrm{par}(x) = \mathrm{par}(y)$. By Fact 6.4.25, both $\{x, \mathrm{par}(x)\}$ and $\{y, \mathrm{par}(y)\}$ appear in $\widehat{p}$ which means that $\mathrm{par}(x)$ appears twice in $\widehat{p}$. By Fact 6.4.21, $\{\mathrm{par}(x), \mathrm{par}(\mathrm{par}(x))\}$ is a redundant edge. Since $u = \mathrm{par}(\mathrm{par}(\cdots \mathrm{par}(x)))$, $\{u, \mathrm{par}(u)\}$ must be a redundant edge according to Fact 6.4.22. □

By Fact 6.4.25 and Fact 6.4.26, we have

$$w(\widetilde{p}) \leq \sum_{e \in E : e \text{ is crucial}} w(e) + 2 \sum_{e \in E : e \text{ is redundant}} w(e). \tag{6.16}$$

Let $\widetilde{p}'' = (\widetilde{u}''_0, \widetilde{u}''_1, \widetilde{u}''_2, \cdots, \widetilde{u}''_{\widetilde{h}''})$. It is obvious that $\widetilde{p}''$ is a path in G'' connecting $\mathrm{root}(s)$ and $\mathrm{root}(t)$. In addition, we have $w(\widetilde{p}) = l(s) + w''(\widetilde{p}'')$. Thus, to conclude,

$$\mathbf{E}\left[l(s) + \mathrm{dist}_{G''}(\mathrm{root}(s), \mathrm{root}(t))\right]$$

$$
\begin{aligned}
&\leq \mathbf{E}[l(s) + w''(\widetilde{p}'')] \\
&\leq \mathbf{E}\left[\sum_{e\in E: e \text{ is crucial}} w(e) + 2 \sum_{e\in E: e \text{ is redundant}} w(e)\right] \\
&\leq \mathbf{E}[w(\widehat{p})] + \epsilon \cdot \mathrm{dist}_G(s,t) \\
&= \left(\sum_{\{u,v\}\in E} w(u,v) \cdot |f(u,c)|\right) + \epsilon \cdot \mathrm{dist}_G(s,t) \\
&\leq (1+2\epsilon)\,\mathrm{dist}_G(s,t),
\end{aligned}
$$

where the first step follows from that $\widetilde{p}''$ is a path in G'' connecting root(s) and root(t), the second step follows from Equation 6.16, the third step follows from Fact 6.4.23, the forth step follows from Lemma 6.4.14, and the last step follows from that f is a $(1+\epsilon)$-approximate uncapacitated minimum cost flow from s to t. □

As proved in the proof of Lemma 6.4.15, $|V''| \leq \lceil n/2 \rceil$. By induction hypothesis, we have $\mathbf{E}[w''(p'')] \leq (1+2\epsilon)^{\lceil \log n \rceil - 1} \cdot \mathbf{E}[\mathrm{dist}_{G''}(\mathrm{root}(s), \mathrm{root}(t))]$. Thus, we have

$$
\begin{aligned}
\mathbf{E}[w(p)] &\leq \mathbf{E}[w''(p'') + l(s)] \\
&\leq (1+2\epsilon)^{\lceil \log n \rceil - 1}\, \mathbf{E}[\mathrm{dist}_{G''}(\mathrm{root}(s), \mathrm{root}(t))] + \mathbf{E}[l(s)] \\
&\leq (1+2\epsilon)^{\lceil \log n \rceil - 1}\, \mathbf{E}[\mathrm{dist}_{G''}(\mathrm{root}(s), \mathrm{root}(t)) + l(s)] \\
&\leq (1+2\epsilon)^{\lceil \log n \rceil}\, \mathrm{dist}_G(s,t),
\end{aligned}
$$

where the first step follows from Claim 6.4.17, the second step follows from induction hypothesis, the last step follows from Claim 6.4.18. □

Theorem 6.4.27 (Parallel approximate $s-t$ shortest path)**.** *Given an $\epsilon \in (0, 0.5)$, a connected n-vertex m-edge undirected graph $G = (V, E, w)$ with $w : E \to \mathbb{Z}_{\geq 0}$, and two vertices $s, t \in V$, there is a PRAM algorithm which takes $\epsilon^{-2}\,\mathrm{poly}(\log(n\Lambda))$ parallel time using expected $\epsilon^{-3} m\,\mathrm{poly}(\log(n\Lambda))$ work and with probability at least 0.99 outputs an $s-t$ path satisfying $w(p) \leq (1+\epsilon) \cdot \mathrm{dist}_G(s,t)$.*

Proof. We invoke FINDPATH(G, s, t, ϵ') $\Theta(\epsilon^{-1} \log n)$ times, where $\epsilon' = \frac{\epsilon}{20 \log n}$. The depth and work is shown by Lemma 6.4.15. As mentioned in the proof of Lemma 6.4.15, we can repeat line 5 $\Theta(\log n)$ times to boost the success probability of computing the flow to $1 - n^{-10}$. By taking union bound, all the flow computation succeed with probability at least 0.999. Condition on success of all the flow computation, by Lemma 6.4.16, FINDPATH(G, s, t, ϵ') outputs a path p satisfies $\mathbf{E}[w(p)] \leq (1 + 2\epsilon')^{\lceil \log n \rceil} \cdot \text{dist}_G(s, t) \leq (1 + \epsilon/2) \cdot \text{dist}_G(s, t)$. By repeating $\Theta(\epsilon^{-1} \log n)$ times, with probability at least 0.999, we can find an $s - t$ path p such that $w(p) \leq (1 + \epsilon) \text{dist}_G(s, t)$. □

6.4.6 Parallel approximate single source shortest paths

In this section, we show how to extend the idea from the previous section to compute approximate single source shortest paths in parallel.

Dual solution for uncapacitated minimum cost flow

Firstly, let us introduce an additional tool, the dual solution for uncapacitated minimum cost flow. Given an undirected graph $G = (V, E, w)$ with $|V| = n$ vertices and $|E| = m$ edges, let $A \in \mathbb{R}^{n \times m}$ be the corresponding vertex-edge incidence matrix, and let diagonal matrix $W \in \mathbb{R}^{m \times m}$ be the corresponding weight matrix. Given a demand vector $b \in \mathbb{R}^n$ with $\mathbf{1}_n^\top b = 0$, i.e., $\sum_{i=1}^n b_i = 0$, the dual problem of the uncapacitated minimum cost flow (transshipment) problem is to solve the following problem:

$$\max_{\zeta \in \mathbb{R}^n} \zeta^\top b$$
$$s.t. \ \ \|\zeta^\top A W^{-1}\|_\infty \leq 1.$$

In other words, in the dual problem, the goal is to find potentials $\zeta \in \mathbb{R}^n$ of vertices that satisfies $\forall e = \{i, j\} \in E, |\zeta_i - \zeta_j| \leq w(e)$ to maximize $\sum_{i=1}^n b_i \cdot \zeta_i$. If a potential vector $\zeta \in \mathbb{R}^n$ satisfies $\|\zeta^\top W^{-1} A\|_\infty \leq 1$, then we say ζ is a feasible dual solution. Notice that $\mathbf{0}_n$ is always a feasible dual solution. Let $\zeta^* \in \mathbb{R}^n$ be the optimal solution for the dual. If a feasible dual solution ζ satisfies that

$\zeta^\top b \geq (1-\epsilon) \cdot \zeta^{*\top} b$, then ζ is a $(1-\epsilon)$-approximate dual solution.

Let x^* be the optimal primal solution of the uncapacitated minimum cost flow problem (the optimal solution of Equation (6.8)). By duality of the uncapacitated minimum cost flow problem (see e.g., the textbook [99]), we know that $\zeta^{*\top} b = \|x^*\|_1$, and thus for any feasible dual solution $\zeta \in \mathbb{R}^n$ and any feasible primal solution $x \in \mathbb{R}^m$, we have $\zeta^\top b \leq \|x\|_1$.

By using the same multiplicative weights update algorithm [66, 67], Algorithm 38, we are able to find an approximate solution for the dual of the transshipment problem. This is also observed by [100].

Theorem 6.4.28 (Parallel algorithm for the dual of uncapacitated minimum cost flow)**.** *Given an $\epsilon \in (0, 0.5)$, a connected n-vertex m-edge undirected weighted graph $G = (V, E, w)$ with $w : E \to Z_{\geq 0}$, and a demand vector $b \in \mathbb{R}^n$ with $\mathbf{1}_n^\top b = 0$, there is a PRAM algorithm which outputs an $(1-\epsilon)$-approximate dual solution to the uncapacitated minimum cost flow problem with probability at least 0.99. Furthermore, the depth is at most $\epsilon^{-2} \log^{O(1)}(n\Lambda)$ and the expected work is at most $\epsilon^{-2} m \cdot \log^{O(1)}(n\Lambda)$, where $\Lambda = \sum_{e \in E} w(e)$.*

Proof. Let $A \in \mathbb{R}^{n \times m}$ be the vertex-edge incidence matrix of G, and let $W \in \mathbb{R}^{m \times m}$ be the weight matrix. As shown in the proof of Theorem 6.4.13, with probability at least 0.99, we can use $\log^{O(1)}(n\Lambda)$ depth and $\widetilde{O}(m)$ work to compute a compressed representation $I = \{I_1, I_2, \cdots, I_n\}$ of a matrix P such that $\kappa(PAW^{-1}) \leq \log^{O(1)}(n\Lambda)$ and $\forall i \in [n], |I_i| \leq \log^{O(1)}(n\Lambda)$ and thus we can implement Algorithm 38 in parallel. In particular, for any $s \geq 1$ and any $\kappa = \log^{O(1)}(n\Lambda) \geq \kappa(PAW^{-1})$, $\mathrm{MWU}(P, A, W, b, s, \epsilon, \kappa)$ (see Algorithm 38) can be done in $\epsilon^{-2} \cdot \log^{O(1)}(n\Lambda)$ depth and $\epsilon^{-2} \cdot m \log^{O(1)}(n\Lambda)$ work.

Consider the procedure $\mathrm{MWU}(P, A, W, b, s, \epsilon, \kappa)$ and let $y_1, y_2, \cdots, y_T \in \mathbb{R}^r$ be the same vectors described in Algorithm 38. Let $\bar{y} = \frac{1}{T} \cdot \sum_{t=1}^T y_t$. According to Lemma 6.3.8, if $\mathrm{MWU}(P, A, W, b, s, \epsilon, \kappa)$ returns FAIL, we have

$$\forall j \in [m], \quad \frac{1}{s} \cdot \frac{\bar{y}^\top Pb}{\|Pb\|_1} < \frac{\bar{y}^\top (PAW^{-1})_j}{\|PAW^{-1}\|_{1 \to 1}}, \quad \frac{1}{s} \cdot \frac{\bar{y}^\top Pb}{\|Pb\|_1} < -\frac{\bar{y}^\top (PAW^{-1})_j}{\|PAW^{-1}\|_{1 \to 1}}.$$

Thus, we have:

$$\forall j \in [m], \left| \frac{\bar{y}^\top (PAW^{-1})_j}{\|PAW^{-1}\|_{1\to 1}} \right| < -\frac{1}{s} \cdot \frac{\bar{y}^\top Pb}{\|Pb\|_1}$$

which implies that

$$\left\| (-P^\top \bar{y})^\top AW^{-1} \right\|_\infty < \frac{(-P^\top \bar{y})^\top b}{s \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}}}.$$

If we set

$$\zeta = s \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}} \cdot \frac{1}{(-P^\top \bar{y})^\top b} \cdot (-P^\top \bar{y}),$$

we have $\|\zeta^\top AW^{-1}\|_\infty < 1$ and $\zeta^\top b = s \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}}$. Thus, we can obtain a feasible dual solution ζ with objective value $s \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}}$. To compute ζ, we need to compute $\|Pb\|_1$, $\|PAW^{-1}\|_{1\to 1}$ and $P^\top \bar{y}$. As discussed in the proof of Theorem 6.4.13, $\|Pb\|_1$ and $\|PAW^{-1}\|_{1\to 1}$ can be computed in $\log^{O(1)}(n\Lambda)$ depth and $m \cdot \log^{O(1)}(n\Lambda)$ work. Notice that $P^\top \bar{y} = \frac{1}{T} \sum_{t=1}^{T} P^\top y_t$ and $\forall t \in [T], P^\top y_t$ can be computed in $\log^{O(1)}(n\Lambda)$ depth and $n \cdot \log^{O(1)}(n\Lambda)$ work as discussed in the proof of Theorem 6.4.13. Since $T = O(\epsilon^{-2}\kappa^2 \log n) = \epsilon^{-2} \log^{O(1)}(n\Lambda)$, $P^\top \bar{y}$ can be computed in $\epsilon^{-2} \log^{O(1)}(n\Lambda)$ depth and $\epsilon^{-2} n \cdot \log^{O(1)}(n\Lambda)$ work. Overall, ζ can be computed in $\epsilon^{-2} \log^{O(1)}(n\Lambda)$ depth and $\epsilon^{-2} m \cdot \log^{O(1)}(n\Lambda)$ total work.

Follow the binary search procedure (see Section 6.3.1), we take $\log(\epsilon^{-1} \log \kappa)$ rounds to find $\widehat{s} \in \{1, 1+\epsilon, (1+\epsilon)^2, \cdots, (1+\epsilon)^{\lceil \log_{1+\epsilon} \kappa \rceil}\}$, we can compute a feasible dual solution ζ satisfying $\zeta^\top b \geq \widehat{s} \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}}$ and according to Lemma 6.3.8, we can also obtain $x \in \mathbb{R}^m$ such that $\|x\|_1 \leq (1+\epsilon) \cdot \widehat{s} \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}}$ and $\|PAW^{-1}x - Pb\|_1 \leq \frac{\epsilon}{\kappa} \cdot \|PAW^{-1}\|_{1\to 1} \cdot (1+\epsilon) \cdot \widehat{s} \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}}$. The overall depth is $O(\epsilon^{-2} \log^{O(1)}(n\Lambda))$ and the overall work is $O(\epsilon^{-2} m \log^{O(1)}(n\Lambda))$. The remaining thing is to show that $\zeta^\top b$ is nearly optimal.

Consider $\widetilde{x} \in \mathbb{R}^m$ with $PAW^{-1}\widetilde{x} = Pb - PAW^{-1}x$ that $\|\widetilde{x}\|_1$ is minimized. We have $\|\widetilde{x}\|_1 \leq \kappa(PAW^{-1}) \cdot \|PAW^{-1}x - Pb\|_1 / \|PAW^{-1}\|_{1\to 1} \leq \kappa \cdot \|PAW^{-1}x - Pb\|_1 / \|PAW^{-1}\|_{1\to 1} \leq \epsilon \cdot (1 +$

$\epsilon) \cdot \widehat{s} \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}}$. Let $x^* \in \mathbb{R}^m$ be the optimal primal solution of the original problem, i.e., $PAW^{-1}x^* = Pb$ and $\|x^*\|_1$ is minimized. We have $\|x^*\|_1 \leq \|x\|_1 + \|\widetilde{x}\|_1 \leq (1+\epsilon)\cdot\widehat{s}\cdot\frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}} + \epsilon\cdot (1+\epsilon)\cdot\widehat{s}\cdot\frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}} \leq (1+\epsilon)^2\cdot\widehat{s}\cdot\frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to 1}}$. It implies that $\zeta^\top b \geq \|x^*\|_1/(1+\epsilon)^2 \geq (1-O(\epsilon))\cdot\|x^*\|_1$. Thus ζ is a $(1 - O(\epsilon))$ approximate dual solution. By adjusting ϵ by a constant, we complete the proof. □

Expected single source shortest path tree

In this section, we show how to extend the recursive path construction method described in Section 6.4.5 to the recursive tree construction method for computing an expected single source shortest path tree. To adapt the same presentation as Section 6.4.5, instead of discussing the single source shortest path tree, we discuss the single sink/target shortest path tree in this section. Notice that the single sink shortest path tree is exactly the same problem as the single source shortest path tree in the undirected graph.

Given a vertex $t \in V$, a special case of uncapacitated minimum cost flow is when the demand $b \in \mathbb{R}^n$ satisfies that $b_t \leq 0$, and $\forall u \in V \setminus \{t\}, b_u \geq 0$ and $\sum_{u\in V\setminus\{t\}} b_u = -b_t$. In this case, the value of the minimum cost flow is exactly the same as $\sum_{u\in V\setminus\{t\}} b_u \cdot \mathrm{dist}_G(u,t)$. If $b = \mathbf{0}_n$, then the optimal flow is a zero flow. Otherwise, we can without loss of generality assume $b_t = -1$, since we can always scale the demand vector b to make $b_t = -1$ and scale back the the solution. In the case of $b_t = -1$, the optimal cost can be seen as $\mathbf{E}_{u\in V\setminus\{t\}}[\mathrm{dist}_G(u,t)]$ where each vertex $u \in V \setminus \{t\}$ is drawn with probability b_u. The optimal flow should route exactly on the shortest path tree with sink vertex t. As shown previously, since we can compute a $(1+\epsilon)$-approximation to the uncapacitated minimum cost flow, we can compute a $(1+\epsilon)$-approximation to the expected distance $\mathbf{E}_{u\in V\setminus\{t\}}[\mathrm{dist}_G(u,t)]$ where each vertex $u \in V \setminus \{t\}$ is drawn with probability b_u. Again, since our flow algorithm can only output a flow which satisfies the demand, it is not necessary to be a tree. We show how to obtain a tree such that the cost of routing demands on the tree is a $(1+\epsilon)$-approximation to the expected distance. The algorithm is shown in Algorithm 43, which is a natural extension of Algorithm 42.

Algorithm 43 Finding an Expected Shortest Path Tree

procedure FINDEXPECTEDTREE($G = (V, E, w), t \in V, \epsilon \in (0, 0.5), \widehat{b} \in \mathbb{R}^{|V|} : \mathbf{1}_n^\top \widehat{b} = 0, \widehat{b}_t \le 0, \forall u \in V \setminus \{t\}, \widehat{b}_u \ge 0$)

Output: $G_T = (V, T, w_T)$ ▷ Return the edges of the tree.

If $\widehat{b} = \mathbf{0}_n$, return $G_T = (V, T, w_T)$ which is an arbitrary spanning tree of G. $\forall e \in T, w_T(e) = w(e)$.

$n \leftarrow |V|, m \leftarrow |E|$. Initialize the demand vector $b \leftarrow \widehat{b}/(-\widehat{b}_t)$.

Compute a $(1+\epsilon)$-approximate uncapacitated minimum cost flow f satisfying b. ▷ Theorem 6.4.13.

For each vertex $u \in V \setminus \{t\}$, set the pointer $\mathrm{par}(u) \leftarrow v \in \{v' \in V \mid f(u, v') > 0\}$ with probability

$$\frac{f(u, v)}{\sum_{v': f(u,v')>0} f(u, v')}.$$

Let $G' = (V, E')$, where $E' = \{\{u, \mathrm{par}(u)\} \mid u \in V \setminus \{t\}\}$.

Compute a spanning forest of G'. For $u \in V$, if u is in the same connected component as t, set $\mathrm{root}(u) \leftarrow t$; otherwise set $\mathrm{root}(u) \leftarrow v$ where v is in the same connected component as u and the edge $\{v, \mathrm{par}(v)\}$ does not appear in the spanning forest.

For $u \in V$, compute $l(u) \leftarrow w(u, \mathrm{par}(u)) + w(\mathrm{par}(u), \mathrm{par}(\mathrm{par}(u))) + \cdots + w(\mathrm{par}(\cdots \mathrm{par}(u)), \mathrm{root}(u))$.

Set $V'' \leftarrow \{v \in V \mid \mathrm{root}(v) = v\}, E'' \leftarrow \{\{\mathrm{root}(u), \mathrm{root}(v)\} \mid \{u, v\} \in E\}$.

For each $e'' = \{u'', v''\} \in E''$, set

$$\mathrm{map}(e'') \leftarrow \arg \min_{\substack{\{u,v\} \in E: \\ \mathrm{root}(u)=u'', \mathrm{root}(v)=v''}} l(u) + w(u, v) + l(v).$$

For each $e'' \in E''$, set $w''(e'') \leftarrow l(u) + w(u, v) + l(v)$, where $\{u, v\} = \mathrm{map}(e'')$.

Set demand vector $b'' \in \mathbb{R}^{|V''|}$: for $u \in V'', b''_u \leftarrow \sum_{v \in V: \mathrm{root}(v)=u} b_v$.

$G''_{T''} = (V'', T'', w''_{T''}) \leftarrow$ FINDEXPECTEDTREE$(G'' = (V'', E'', w''), t, \epsilon, b'')$. ▷ Recursion.

$T \leftarrow \{\{u, \mathrm{par}(u)\} \mid u \in V \setminus \{t\}, \mathrm{root}(u) \ne u\} \cup \{\mathrm{map}(e'') \mid e'' \in T''\}$.

For each $e \in T$, set $w_T(e) \leftarrow w(e)$. Return $G_T = (V, T, w_T)$.

end procedure

Lemma 6.4.29 (Work and depth of parallel expected shortest path tree)**.** *Given an $\epsilon \in (0, 0.5)$, a connected n-vertex m-edge undirected graph $G = (V, E, w)$ with $w : E \to \mathbb{Z}_{\ge 0}$, a vertex $t \in V$, and a demand vector $\widehat{b} \in \mathbb{R}^n$ satisfying $\mathbf{1}_n^\top \widehat{b} = 0, \widehat{b}_t \le 0, \forall u \in V \setminus \{t\}, \widehat{b}_u \ge 0$, FINDEXPECTEDTREE$(G, t, \epsilon, \widehat{b})$ (Algorithm 43) can be implemented in PRAM with $\epsilon^{-2} \log^{O(1)}(n\Lambda)$ depth and expected $\epsilon^{-2} m \log^{O(1)}(n\Lambda)$ work, where $\Lambda = \max_{e \in E} w(e)$.*

Proof. The proof is similar to the proof of Lemma 6.4.15. Line 3 can be done in $\log^{O(1)} n$ depth and $m \cdot \log^{O(1)} n$ work (see, e.g., [13]). Line 5 can be done in $\epsilon^{-2} \mathrm{poly}(\log(n\Lambda))$ depth using expected $\epsilon^{-2} m \, \mathrm{poly}(\log(n\Lambda))$ work by Theorem 6.4.13. We can repeat line 5 $\Theta(\log n)$ times to boost the success probability to $1 - n^{-10}$. It only increases the work by a $O(\log n)$ factor. Line 6 can be done

in $\text{poly}(\log n)$ depth using $m\,\text{poly}(\log n)$ work. In line 8, computing connected components and a spanning forest can be done in $\text{poly}(\log n)$ depth using $m\,\text{poly}(\log n)$ work [13]. In line 9, we can use doubling algorithm to compute $l(u)$ for all $u \in V$ simultaneously in $\text{poly}(\log n)$ depth using $n\,\text{poly}(\log n)$ work.

Next, consider the number of recursions (line 14) needed. Except vertex t, every vertex u has an edge $\{u, \text{par}(u)\} \in E'$. Thus, each connected component without t in G' has size at most 2. Notice that $|V''|$ is exactly the same as the number of connected components in G'. Thus, $|V''| \leq \lceil n/2 \rceil$. It implies that the number of recursions is at most $\lceil \log n \rceil$. Thus, the overall depth is at most $\epsilon^{-2}\,\text{poly}(\log(n\Lambda))$ and the overall expected work is at most $\epsilon^{-2} m\,\text{poly}(\log(n\Lambda))$. □

Lemma 6.4.30 (Correctness of parallel approximate expected single sink shortest path tree)**.** *Given an $\epsilon \in (0, 0.5)$, a connected n-vertex m-edge undirected graph $G = (V, E, w)$ with $w : E \to \mathbb{Z}_{\geq 0}$, a vertex $t \in V$ and a demand vector $\widehat{b} \in \mathbb{R}^n$ satisfying $\mathbf{1}_n^\top \widehat{b} = 0, \widehat{b}_t \leq 0, \forall u \in V \setminus \{t\}, \widehat{b}_u \geq 0$, let $G_T = (V, T, w_T)$ be the output of* FINDEXPECTEDTREE$(G, t, \epsilon, \widehat{b})$*. Then G_T is a spanning tree of G and $\mathbf{E}\left[\sum_{u \in V \setminus \{t\}} \widehat{b}_u \cdot \text{dist}_{G_T}(u, t)\right] \leq (1 + 2\epsilon)^{\lceil \log n \rceil} \cdot \sum_{u \in V \setminus \{t\}} \widehat{b}_u \cdot \text{dist}_G(u, t)$.*

Proof. The proof is similar to the proof of Lemma 6.4.16. Our proof is by induction on the number of vertices of G. When G only has one vertex, the statement is obviously true. Now suppose the statement is true for any graph with less than n vertices and consider $G = (V, E, w)$ with $|V| = n$ vertices and $|E| = m$ edges.

For $u \in V$, let $l(u)$ be the same as that in line 9. Let $G_T = (V, T, w_T)$ be the output of FINDEXPECTEDTREE$(G, t, \epsilon, \widehat{b})$. Let $G''_{T''}$ be the output of line 14. Let root(u) be the same as in line 8 for $u \in V$. By line 8, we know that root$(t) = t$ and thus $t \in V''$.

Firstly, we show that G_T is a spanning tree of G. By line 8, we know that $\{\{u, \text{par}(u)\} \mid u \in V \setminus \{t\}, \text{root}(u) \neq u\}$ forms a forest. Each tree in the forest has exactly one root. Thus, the number of edges in the forest is $n - |\{u \in V \mid u = \text{root}(u)\}|$. By induction hypothesis, $G''_{T''}$ is a spanning tree of G''. By line 15, if root(u) and root(v) is connected in $G''_{T''}$, we will add an edge to connect the tree containing root(u) with the tree containing root(v) in G_T. Since $G''_{T''}$ is a spanning tree of G'', any two roots are in the same connected component in G'' which implies that any two vertices

in G are in the same connected component in G_T. Since $|T''| = |\{u \in V \mid u = \text{root}(u)\}| - 1$, we have $|T| = n - |\{u \in V \mid u = \text{root}(u)\}| + |\{u \in V \mid u = \text{root}(u)\}| - 1$. Thus, T has $n-1$ edges which implies that G_T is a spanning tree.

In the remaining of the proof, let us prove that the routing cost on G_T is small.

Claim 6.4.31 (Analogue of Claim 6.4.17)**.**

$$\sum_{u \in V \setminus \{t\}} \widehat{b}_u \cdot \text{dist}_{G_T}(u,t) \leq \sum_{u \in V \setminus \{t\}} \widehat{b}_u \cdot \left(l(u) + \text{dist}_{G''_{T''}}(\text{root}(u), t) \right)$$

Proof. It suffices to show that $\forall u \in V \setminus \{t\}, \text{dist}_{G_T}(u,t) \leq l(u) + \text{dist}_{G''_{T''}}(\text{root}(u), t)$. Consider the shortest path p'' from root(u) to t in the tree $G''_{T''}$. Suppose $p'' = (u''_0, u''_1, \cdots, u''_{h''})$, where $u''_0 = u, u''_{h''} = t$. According to line 15, we know that $\forall e'' \in T'', \text{map}(e'') \in T$ and $\forall v \in V$, if $v \neq \text{root}(v)$, the edge $\{v, \text{par}(v)\}$ is also in T.

Thus, we have

$$\begin{aligned}
&\text{dist}_{G_T}(u,t) \\
\leq & w(u, \text{par}(u), \text{par}(\text{par}(u)), \cdots, \text{root}(u)) \\
&+ \sum_{\substack{x_i, y_i: \\ \{x_i, y_i\} = \text{map}(\{u''_{i-1}, u''_i\})}} w((\text{root}(x_i), \cdots, \text{par}(x_i), x_i, y_i, \text{par}(y_i), \cdots, \text{root}(y_i))) \\
&= l(u) + \sum_{\substack{x_i, y_i: \\ \{x_i, y_i\} = \text{map}(\{u''_{i-1}, u''_i\})}} l(x_i) + w(x_i, y_i) + l(y_i) \\
&= l(u) + \sum_{i=1}^{h''} w''(u''_{i-1}, u''_i) \\
&= l(u) + \text{dist}_{G''_{T''}}(\text{root}(u), t),
\end{aligned}$$

where the first equality follows from line 9 and the second inequality follows from line 12. □

Let demand vector b be the same as in line 4. Let demand vector b'' be the same as in line 13.

Claim 6.4.32.

$$\sum_{u\in V\setminus\{t\}} \widehat{b}_u \cdot \left(l(u) + \mathrm{dist}_{G''_{T''}}(\mathrm{root}(u), t)\right) = (-\widehat{b}_t) \cdot \left(\sum_{u\in V\setminus\{t\}} b_u \cdot l(u) + \sum_{u''\in V''\setminus\{t\}} b''_{u''} \cdot \mathrm{dist}_{G''_{T''}}(u'', t)\right).$$

Proof.

$$\begin{aligned}
&\sum_{u\in V\setminus\{t\}} \widehat{b}_u \cdot \left(l(u) + \mathrm{dist}_{G''_{T''}}(\mathrm{root}(u), t)\right)\\
=&(-\widehat{b}_t) \cdot \left(\sum_{u\in V\setminus\{t\}} b_u \cdot l(u) + \sum_{u\in V\setminus\{t\}} b_u \cdot \mathrm{dist}_{G''_{T''}}(\mathrm{root}(u), t)\right)\\
=&(-\widehat{b}_t) \cdot \left(\sum_{u\in V\setminus\{t\}} b_u \cdot l(u) + \sum_{u\in V} b_u \cdot \mathrm{dist}_{G''_{T''}}(\mathrm{root}(u), t)\right)\\
=&(-\widehat{b}_t) \cdot \left(\sum_{u\in V\setminus\{t\}} b_u \cdot l(u) + \sum_{u''\in V''} \sum_{u\in V:\mathrm{root}(u)=u''} b_u \cdot \mathrm{dist}_{G''_{T''}}(u'', t)\right)\\
=&(-\widehat{b}_t) \cdot \left(\sum_{u\in V\setminus\{t\}} b_u \cdot l(u) + \sum_{u''\in V''} b''_{u''} \cdot \mathrm{dist}_{G''_{T''}}(u'', t)\right),
\end{aligned}$$

where the first step follows from that $b_u = \widehat{b}_u/(-\widehat{b}_t)$, the second step follows from that $\mathrm{root}(t) = t$, the third step follows from that $V'' = \{\mathrm{root}(u) \mid u \in V\}$ and $\mathrm{dist}_{G''_{T''}}(t, t) = 0$ and the last step follows from that $b''_{u''} = \sum_{u\in V:\mathrm{root}(u)=u''} b_u$ (see line 13). □

Claim 6.4.33 (Analogue of Claim 6.4.18)**.**

$$\mathop{\mathbf{E}}_{\mathrm{par}:V\setminus\{t\}\to V}\left[\sum_{u\in V\setminus\{t\}} b_u \cdot l(u) + \sum_{u''\in V''\setminus\{t\}} b''_{u''} \cdot \mathrm{dist}_{G''}(u'', t)\right] \le (1+2\epsilon) \cdot \sum_{u\in V\setminus\{t\}} b_u \cdot \mathrm{dist}_G(u, t).$$

Proof. The proof is similar to the proof of Claim 6.4.18. Let f be the flow obtained by line 5 satisfying $\sum_{\{u,v\}\in E} w(u, v) \cdot |f(u, v)| \le (1+\epsilon) \sum_{u\in V\setminus\{t\}} b_u \cdot \mathrm{dist}_G(u, t)$. Our proof is by coupling. Notice that $b_t = -1$ and $\forall u \in V \setminus \{t\}, b_u \ge 0$. By Lemma 6.4.14, we only need to prove that $\mathbf{E}_{\mathrm{par}:V\setminus\{t\}\to V}\left[\sum_{u\in V\setminus\{t\}} b_u \cdot l(u) + \sum_{u''\in V''\setminus\{t\}} b''_{u''} \cdot \mathrm{dist}_{G''}(u'', t)\right]$ is almost bounded by the expected

length of a random walk corresponding to the flow f.

For $u \in V$, let $\mathrm{par}(u)$ be the same as that in Algorithm 43. We conceptually generate a random walk $\widehat{p}$ corresponding to f in the following way:

1. Set $i \leftarrow 0$, and set $\widehat{u}_0$ to be $v \in V \setminus \{t\}$ with probability b_v.
2. If $\forall j \in \{0, 1, \cdots, i-1\}, \widehat{u}_j \neq \widehat{u}_i$, then set $\widehat{u}_{i+1} \leftarrow \mathrm{par}(\widehat{u}_i)$. Otherwise, set $\widehat{u}_{i+1}$ to be $v \in \{v' \in V \mid f(\widehat{u}_i, v') > 0\}$ with probability $\frac{f(\widehat{u}_i, v)}{\sum_{v': f(\widehat{u}_i, v') > 0} f(\widehat{u}_i, v')}$.
3. If $\widehat{u}_{i+1} \neq t$, set $i \leftarrow i + 1$, and repeat step 2 . Otherwise, $\widehat{h} \leftarrow i + 1$, and output the path $\widehat{p} = (\widehat{u}_0, \widehat{u}_1, \cdots, \widehat{u}_{\widehat{h}})$ as the random walk.

It is easy to see that $\widehat{p}$ is a random walk corresponding to the flow f and thus $\mathbf{E}[w(\widehat{p})] = \sum_{\{u,v\} \in E} w(u, v) \cdot |f(u, v)|$ by Lemma 6.4.14. Notice that $\widehat{p}$ may have cycles, we conceptually do the following procedure to shortcut all cycles of $\widehat{p}$ to obtain $\bar{p}$:

1. Set $i \leftarrow 0, j \leftarrow 0$.
2. If $\nexists i' > i, \widehat{u}_{i'} = \widehat{u}_i$, set $\bar{u}_j \leftarrow \widehat{u}_i, j \leftarrow j + 1, i \leftarrow i + 1$.
3. Otherwise, find the largest $i' \in [\widehat{h}]$ such that $\widehat{u}_{i'} = \widehat{u}_i$. Mark all edges $\{\widehat{u}_i, \widehat{u}_{i+1}\}$, $\{\widehat{u}_{i+1}, \widehat{u}_{i+2}\}$, $\cdots$, $\{\widehat{u}_{i'-1}, \widehat{u}_{i'}\}$ as *redundant*.
4. Repeat above two steps until $i > \widehat{h}$. At the end, set $\bar{h} \leftarrow j - 1$ and path $\bar{p} \leftarrow (\bar{u}_0, \bar{u}_1, \cdots, \bar{u}_{\bar{h}})$. For each edge $e \in E$ which appears in $\widehat{p}$ but is not marked as redundant, mark e as *crucial*.

It is easy to see that $\bar{p}$ is a simple s-t path and furthermore $\sum_{e \in E: e \text{ is crucial}} w(e) \leq w(\bar{p}) \leq w(\widehat{p}) - \sum_{e \in E: e \text{ is redundant}} w(e)$.

By the above constructions, $\widehat{p}$ has several good properties.

Fact 6.4.34 (The same as Fact 6.4.19)**.** *If $\widehat{u}_i$ is the first appearance of a vertex v, then $\widehat{u}_{i+1}$ must be* $\mathrm{par}(v)$.

Fact 6.4.35 (The same as Fact 6.4.20)**.** *If vertex $v \in V$ appears in $\widehat{p}$ and $v \neq t$, all edges* $\{v, \mathrm{par}(v)\}$, $\{\mathrm{par}(v), \mathrm{par}(\mathrm{par}(v))\}$, $\{\mathrm{par}(\mathrm{par}(v)), \mathrm{par}(\mathrm{par}(\mathrm{par}(v)))\}, \cdots$ *should be in $\widehat{p}$.*

Proof. It follows from Fact 6.4.34. □

Fact 6.4.36 (The same as Fact 6.4.21)**.** *If vertex v appears at least twice in $\widehat{p}$, $\{v, \mathrm{par}(v)\}$ is a redundant edge.*

Proof. The proof is exactly the same as the proof of Fact 6.4.21.

□

Fact 6.4.37 (The same as Fact 6.4.22)**.** *If $\{v, \mathrm{par}(v)\}$ is a redundant edge, $\{\mathrm{par}(v), \mathrm{par}(\mathrm{par}(v))\}$ is a redundant edge.*

Proof. The proof is exactly the same as the proof of Fact 6.4.22. □

Fact 6.4.38 (Analogue of Fact 6.4.23)**.**

$$\mathop{\mathbf{E}}_{\widehat{p}}\left[\sum_{e\in E: e \text{ is redundant}} w(e)\right] \leq \epsilon \cdot \sum_{u\in V\setminus\{t\}} b_u \cdot \mathrm{dist}_G(u,t).$$

Proof. Condition on that the starting vertex $\widehat{u}_0 = u$, we have

$$\sum_{e\in E: e \text{ is redundant}} w(e) \leq w(\widehat{p}) - w(\bar{p}) \leq w(\widehat{p}) - \mathrm{dist}_G(u,t), \tag{6.17}$$

where the last inequality follows from that $w(\bar{p}) \geq \mathrm{dist}_G(u,t)$. On the other hand, we have:

$$\mathbf{E}[w(\widehat{p})] = \sum_{\{v,v'\}\in E} w(v,v') \cdot |f(v,v')| \leq (1+\epsilon) \sum_{u\in V\setminus\{t\}} b_u \cdot \mathrm{dist}_G(u,t), \tag{6.18}$$

where the first equality follows from Lemma 6.4.14 and the last inequality follows from that f is a $(1+\epsilon)$-approximate uncapacitated minimum cost flow satisfying demands b. Thus,

$$\begin{aligned}
&\mathop{\mathbf{E}}_{\widehat{p}}\left[\sum_{e\in E: e \text{ is redundant}} w(e)\right] \\
&= \sum_{u\in V\setminus\{t\}} \mathop{\mathbf{E}}_{\widehat{p}}\left[\sum_{e\in E: e \text{ is redundant}} w(e) \,\middle|\, \widehat{u}_0 = u\right] \cdot \Pr[\widehat{u}_0 = u]
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{u \in V \setminus \{t\}} \mathop{\mathbf{E}}_{\widehat{p}} \left[\sum_{e \in E : e \text{ is redundant}} w(e) \,\middle|\, \widehat{u}_0 = u \right] \cdot b_u \\
&\leq \sum_{u \in V \setminus \{t\}} b_u \cdot \left(\mathop{\mathbf{E}}_{\widehat{p}} [w(\widehat{p}) \mid \widehat{u}_0 = u] - \mathrm{dist}_G(u, t) \right) \\
&= \sum_{u \in V \setminus \{t\}} \Pr[\widehat{u}_0 = u] \cdot \mathop{\mathbf{E}}_{\widehat{p}} [w(\widehat{p}) \mid \widehat{u}_0 = u] - \sum_{u \in V \setminus \{t\}} b_u \cdot \mathrm{dist}_G(u, t) \\
&= \mathop{\mathbf{E}}_{\widehat{p}} [w(\widehat{p})] - \sum_{u \in V \setminus \{t\}} b_u \cdot \mathrm{dist}_G(u, t) \\
&\leq \epsilon \cdot \sum_{u \in V \setminus \{t\}} b_u \cdot \mathrm{dist}_G(u, t),
\end{aligned}
$$

where the second equality follows from that the probability that the start vertex of $\widehat{p}$ is u is b_u, the first inequality follows from Equation (6.17), the third equality again follows from that the probability that the start vertex of $\widehat{p}$ is u is b_u, and the last inequality follows from Equation (6.18).

□

Next we show how to find a path $\widetilde{p}''$ in G'' between $\mathrm{root}(\widehat{u}_0)$ and t such that $l(\widehat{u}_0) + w''(\widetilde{p}'')$ is at most $\sum_{e \in E : e \text{ is crucial}} w(e) + 2 \sum_{e \in E : e \text{ is redundant}} w(e)$. Let us conceptually construct a path $\widetilde{p}$ in G between $\widehat{u}_0$ and t using the following way:

1. Initialize $\widetilde{p} = (\widehat{u}_0, \mathrm{par}(\widehat{u}_0), \mathrm{par}(\mathrm{par}(\widehat{u}_0)), \cdots, \mathrm{root}(\widehat{u}_0)), i \leftarrow 0$. Let $\widetilde{u}''_0 \leftarrow \mathrm{root}(\widehat{u}_0)$.

2. Let $\widehat{u}_{k_i}$ be the last vertex in $\widehat{p}$ such that $\widehat{u}_{k_i}$ is in the same connected component as $\widetilde{u}''_i$ in G' (G' is constructed by line 7).

3. If $\widetilde{u}''_i = t$, let $\widetilde{h}'' \leftarrow i$ and finish the procedure.

4. Otherwise, concatenate

   $$(\mathrm{root}(\widehat{u}_{k_i}) = \widetilde{u}''_i, \cdots, \mathrm{par}(\mathrm{par}(\widehat{u}_{k_i})), \mathrm{par}(\widehat{u}_{k_i}), \widehat{u}_{k_i}, \widehat{u}_{k_i+1}, \mathrm{par}(\widehat{u}_{k_i+1}), \mathrm{par}(\mathrm{par}(\widehat{u}_{k_i+1})), \cdots, \mathrm{root}(\widehat{u}_{k_i+1}))$$

   to $\widetilde{p}$ and set $\widetilde{u}''_{i+1} \leftarrow \mathrm{root}(\widehat{u}_{k_i+1})$.

5. Set $i \leftarrow i + 1$ and go to step 3.

Fact 6.4.39 (The same as Fact 6.4.24)**.** $\forall i \neq j \in \{0, 1, \cdots, \widetilde{h}''\}, \widetilde{u}''_i \neq \widetilde{u}''_j$.

Proof. The proof is exactly the same as Fact 6.4.24. □

Fact 6.4.40 (Analogue of Fact 6.4.25)**.** *Each edge in $\widetilde{p}$ appears in $\widehat{p}$.*

Proof. Edges $\{\widehat{u}_0, \mathrm{par}(\widehat{u}_0)\}, \{\mathrm{par}(\widehat{u}_0), \mathrm{par}(\mathrm{par}(\widehat{u}_0))\}, \{\mathrm{par}(\mathrm{par}(\widehat{u}_0)), \mathrm{par}(\mathrm{par}(\mathrm{par}(\widehat{u}_0)))\}, \cdots$ must appear in $\widehat{p}$ by Fact 6.4.35. By the choice of $\widehat{u}_{k_i}$ and $\widehat{u}_{k_i+1}$, edge $\{\widehat{u}_{k_i}, \widehat{u}_{k_i+1}\}$ appears in $\widehat{p}$. By Fact 6.4.35 again, edges $\{\widehat{u}_{k_i}, \mathrm{par}(\widehat{u}_{k_i})\}, \{\mathrm{par}(\widehat{u}_{k_i}), \mathrm{par}(\mathrm{par}(\widehat{u}_{k_i}))\}, \{\mathrm{par}(\mathrm{par}(\widehat{u}_{k_i})), \mathrm{par}(\mathrm{par}(\mathrm{par}(\widehat{u}_{k_i})))\}, \cdots$ and edges $\{\widehat{u}_{k_i+1}, \mathrm{par}(\widehat{u}_{k_i+1})\}, \{\mathrm{par}(\widehat{u}_{k_i+1}), \mathrm{par}(\mathrm{par}(\widehat{u}_{k_i+1}))\}, \{\mathrm{par}(\mathrm{par}(\widehat{u}_{k_i+1})), \mathrm{par}(\mathrm{par}(\mathrm{par}(\widehat{u}_{k_i+1})))\}, \cdots$ also appear in $\widehat{p}$. □

Fact 6.4.41 (The same as Fact 6.4.26)**.** *Each edge in $\widetilde{p}$ can appear at most twice. In addition, if an edge e in $\widetilde{p}$ appears twice, e is a redundant edge.*

Proof. The proof is exactly the same as the proof of Fact 6.4.26. □

By Fact 6.4.40 and Fact 6.4.41, we have:

$$w(\widetilde{p}) \leq \sum_{e \in E : e \text{ is crucial}} w(e) + 2 \cdot \sum_{e \in E : e \text{ is redundant}} w(e) \leq w(\widehat{p}) + \sum_{e \in E : e \text{ is redundant}} w(e). \tag{6.19}$$

Let $\widetilde{p}'' = (\widetilde{u}''_0, \widetilde{u}''_1, \widetilde{u}''_2, \cdots, \widetilde{u}''_{\widetilde{h}''})$. By the choice of $\widetilde{u}''_0, \widetilde{u}''_1, \widetilde{u}''_2, \cdots, \widetilde{u}''_{\widetilde{h}''}$, it is obvious that $\widetilde{p}''$ is a path in G'' connecting root $\widehat{u}_0$ and t. In addition, we have $w(\widetilde{p}) = l(\widehat{u}_0) + w''(\widetilde{p}'')$. Thus, to conclude, we have:

$$\begin{aligned}
&\mathop{\mathbf{E}}_{\mathrm{par}}\left[\sum_{u \in V \setminus \{t\}} b_u \cdot l(u) + \sum_{u'' \in V'' \setminus \{t\}} b''_{u''} \cdot \mathrm{dist}_{G''}(u'', t)\right] \\
= &\mathop{\mathbf{E}}_{\mathrm{par}}\left[\sum_{u \in V \setminus \{t\}} b_u \cdot l(u) + \sum_{u'' \in V''} b''_{u''} \cdot \mathrm{dist}_{G''}(u'', t)\right] \\
= &\mathop{\mathbf{E}}_{\mathrm{par}}\left[\sum_{u \in V \setminus \{t\}} b_u \cdot l(u) + \sum_{u'' \in V''} \sum_{u \in V : \mathrm{root}(u) = u''} b_u \cdot \mathrm{dist}_{G''}(u'', t)\right]
\end{aligned}$$

$$
\begin{aligned}
&= \mathop{\mathbf{E}}_{\mathrm{par}}\left[\sum_{u\in V\setminus\{t\}} b_u\cdot(l(u)+\mathrm{dist}_{G''}(\mathrm{root}(u),t))\right]\\
&= \sum_{u\in V\setminus\{t\}} b_u\cdot \mathop{\mathbf{E}}_{\mathrm{par}}[l(u)+\mathrm{dist}_{G''}(\mathrm{root}(u),t)]\\
&= \sum_{u\in V\setminus\{t\}} b_u\cdot \mathop{\mathbf{E}}_{\mathrm{par},\widehat{u}_0}[l(u)+\mathrm{dist}_{G''}(\mathrm{root}(u),t)]\\
&= \sum_{u\in V\setminus\{t\}} \Pr_{\widehat{u}_0}[\widehat{u}_0=u]\cdot \mathop{\mathbf{E}}_{\mathrm{par},\widehat{u}_0}[l(\widehat{u}_0)+\mathrm{dist}_{G''}(\mathrm{root}(\widehat{u}_0),t)\mid \widehat{u}_0=u]\\
&= \mathop{\mathbf{E}}_{\mathrm{par},\widehat{u}_0}[l(\widehat{u}_0)+\mathrm{dist}_{G''}(\mathrm{root}(\widehat{u}_0),t)]\\
&\le \mathop{\mathbf{E}}_{\mathrm{par},\widehat{u}_0,\widehat{p}}[l(\widehat{u}_0)+w''(\widehat{p}'')]\\
&= \mathop{\mathbf{E}}_{\mathrm{par},\widehat{u}_0,\widehat{p}}[w(\widehat{p})]\\
&\le \mathop{\mathbf{E}}_{\mathrm{par},\widehat{u}_0,\widehat{p}}[w(\widehat{p})]+\mathop{\mathbf{E}}_{\mathrm{par},\widehat{u}_0,\widehat{p}}\left[\sum_{e\in E:e\text{ is redundant}} w(e)\right]\\
&\le \mathop{\mathbf{E}}_{\mathrm{par},\widehat{u}_0,\widehat{p}}[w(\widehat{p})]+\epsilon\cdot\sum_{u\in V\setminus\{t\}} b_u\cdot\mathrm{dist}_G(u,t)\\
&\le (1+2\epsilon)\cdot\sum_{u\in V\setminus\{t\}} b_u\cdot\mathrm{dist}_G(u,t),
\end{aligned}
$$

where the first step follows from $\mathrm{dist}_{G''}(t,t)=0$, the second step follows from $\forall u''\in V'', b''_{u''}=\sum_{u\in V:\mathrm{root}(u)=u''} b_u$, the third step follows from $V''=\{\mathrm{root}(u)\mid u\in V\}$, the fourth step follows from the linearity of expectation, the fifth step follows from that the choice of $\widehat{u}_0$ does not influence $l(u)+\mathrm{dist}_{G''}(\mathrm{root}(u),t)$, the sixth step follows from that $\Pr[\widehat{u}_0=u]=b_u$, the seventh step follows from conditional expectation, the eighth step follows from that $\widetilde{p}''$ is a path between root $\widehat{u}_0$ and t in G'', the ninth step follows from $w(\widetilde{p})=l(\widehat{u}_0)+w''(\widetilde{p}'')$, the tenth step follows from Equation (6.19), the eleventh step follows from Fact 6.4.38 and the last step follows from Lemma 6.4.14. □

As proved in the proof of Lemma 6.4.29, $|V''|\le\lceil n/2\rceil$. By induction hypothesis, we have $\mathbf{E}[\sum_{u''\in V''\setminus\{t\}} b''_{u''}\cdot\mathrm{dist}_{G''_{T''}}(u'',t)]\le(1+2\epsilon)^{\lceil\log n\rceil-1}\cdot\mathbf{E}_{\mathrm{par}}[\sum_{u\in V\setminus\{t\}} b''_{u''}\,\mathrm{dist}_{G''}(u'',t)]$. Thus, we

have:

$$
\begin{aligned}
&\mathbf{E}\left[\sum_{u\in V\setminus\{t\}}\widehat{b}_u\cdot\mathrm{dist}_{G_T}(u,t)\right]\\
\leq&\mathbf{E}\left[\sum_{u\in V\setminus\{t\}}\widehat{b}_u\cdot\left(l(u)+\mathrm{dist}_{G''_{T''}}(\mathrm{root}(u),t)\right)\right]\\
=&(-\widehat{b}_t)\cdot\mathbf{E}\left[\sum_{u\in V\setminus\{t\}}b_u\cdot l(u)+\sum_{u''\in V''\setminus\{t\}}b''_{u''}\cdot\mathrm{dist}_{G''_{T''}}(u'',t)\right]\\
\leq&(-\widehat{b}_t)\cdot\mathbf{E}\left[\sum_{u\in V\setminus\{t\}}b_u\cdot l(u)\right]+(1+2\epsilon)^{\lceil\log n\rceil-1}\cdot(-\widehat{b}_t)\cdot\mathbf{E}\left[\sum_{u''\in V''\setminus\{t\}}b''_{u''}\cdot\mathrm{dist}_{G''}(u'',t)\right]\\
\leq&(1+2\epsilon)^{\lceil\log n\rceil-1}\cdot(-\widehat{b}_t)\cdot\mathbf{E}\left[\sum_{u\in V\setminus\{t\}}b_u\cdot l(u)+\sum_{u''\in V''\setminus\{t\}}b''_{u''}\cdot\mathrm{dist}_{G''}(u'',t)\right]\\
\leq&(1+2\epsilon)^{\lceil\log n\rceil-1}\cdot(-\widehat{b}_t)\cdot(1+2\epsilon)\cdot\sum_{u\in V\setminus\{t\}}b_u\cdot\mathrm{dist}_G(u,t)\\
\leq&(1+2\epsilon)^{\lceil\log n\rceil}\cdot\sum_{u\in V\setminus\{t\}}\widehat{b}_u\cdot\mathrm{dist}_G(u,t),
\end{aligned}
$$

where the first step follows from Claim 6.4.31, the second step follows from Claim 6.4.32, the third step follows from induction hypothesis, the fourth step follows from $(1+2\epsilon)>1$, the fifth step follows from Claim 6.4.33, and the last step follows from $\widehat{b}=b\cdot(-\widehat{b}_t)$. □

Theorem 6.4.42 (Parallel approximate expected single source (sink) shortest path tree)**.** *Given an $\epsilon\in(0,0.5)$, a connected n-vertex m-edge undirected graph $G=(V,E,w)$ with $w:E\rightarrow\mathbb{Z}_{\geq 0}$, a vertex s, and a demand vector $b\in\mathbb{R}^n$ satisfying $\mathbf{1}_n^\top b=0, b_s\leq 0, \forall u\in V\setminus\{s\}, b_u\geq 0$, there is a PRAM algorithm which takes $\epsilon^{-2}\,\mathrm{poly}(\log(n\Lambda))$ parallel time using expected $\epsilon^{-3}m\,\mathrm{poly}(\log(n\Lambda))$ work and with probability at least 0.99 outputs a spanning tree $G_T=(V,T,w_T)$ of G satisfying $\sum_{u\in V\setminus\{s\}}b_u\cdot\mathrm{dist}_{G_T}(s,u)\leq(1+\epsilon)\cdot\sum_{u\in V\setminus\{s\}}b_u\cdot\mathrm{dist}_G(s,u)$.*

Proof. We invoke FINDEXPECTEDTREE(G,s,ϵ',b) (Algorithm 43) $\Theta(\epsilon^{-1}\log n)$ times, where $\epsilon'=\frac{\epsilon}{20\log n}$. The depth and work is shown by Lemma 6.4.29. As mentioned in the proof of Lemma 6.4.29, we can repeat line 5 $\Theta(\log n)$ times to boost the success probability of computing the flow to

$1 - n^{-10}$. By taking union bound, all the flow computation succeed with probability at least 0.999. Condition on success of all the flow computation, by Lemma 6.4.30, FINDEXPECTEDTREE(G, s, ϵ', b) outputs a spanning tree $G_T = (V, T, w_T)$ of G and $\mathbf{E}[\sum_{u \in V \setminus \{s\}} b_u \cdot G_T(s,u)] \leq (1 + 2\epsilon')^{\lceil \log n \rceil} \cdot \sum_{u \in V \setminus \{s\}} b_u \cdot \text{dist}_G(s,u) \leq (1 + \epsilon/2) \cdot \sum_{u \in V \setminus \{s\}} b_u \cdot \text{dist}_G(s,u)$. By repeating $\Theta(\epsilon^{-1} \log n)$ times in parallel, with probability at least 0.999, we can find a spanning tree G_T such that $\sum_{u \in V \setminus \{s\}} b_u \cdot \text{dist}_{G_T}(s,u) \leq (1 + \epsilon) \cdot \sum_{u \in V \setminus \{s\}} b_u \cdot \text{dist}_G(s,u)$. □

Parallel approximate single source shortest path tree and potentials

We can use Theorem 6.4.28 to compute an approximate dual solution for expected single source shortest path problem. We can use Theorem 6.4.42 to compute an approximate (primal) solution for expected single source shortest path problem. In [43], they show how to use primal and dual expected solution to compute approximate single source shortest paths. [100] also adapted and extended these ideas to compute approximate single source shortest path tree and approximate single source shortest path potentials in parallel setting. In this section, we present this method for completeness.

The following lemma shows how to implement line 11.

Lemma 6.4.43 (Merge two spanning trees in parallel)**.** *Given two trees $G_T = (V, T, w_T)$ with weights $w_T : T \to \mathbb{Z}_{\geq 0}$, $G_{T'} = (V, T', w_{T'})$ with weights $w_{T'} : T' \to \mathbb{Z}_{\geq 0}$ on the same set of n vertices V and a vertex $s \in V$, there is a PRAM algorithm which outputs a tree $G_{T''} = (V, T'', w_{T''})$ on V in $\text{poly}(\log n)$ depth and $n \cdot \text{poly}(\log n)$ work such that $\forall u \in V, \text{dist}_{G_{T''}}(s,u) \leq \min(\text{dist}_{G_T}(s,u), \text{dist}_{G_{T'}}(s,u))$. Furthermore, $T'' \subseteq T \cup T'$ and $\forall e \in T''$, either $w_{T''}(e) = w_T(e)$ or $w_{T''}(e) = w_{T'}(e)$.*

Proof. Firstly, we can without loss of generality assume that all weights are positive. Otherwise, let $\Lambda = \max(\max_{e \in T} w_T(e), \max_{e \in T'} w_{T'}(e'))$, and scale the weights: $\forall e \in T, w_T(e) \leftarrow w_T(e) \cdot n \cdot (\Lambda + 1) + 1, \forall e \in T', w_{T'}(e) \leftarrow w_{T'}(e) \cdot n \cdot (\Lambda + 1) + 1$. Then all weights become positive, and it is easy to see that $\forall u \in V \setminus \{s\}, \lfloor \text{dist}_{G_T}(s,u)/(n \cdot (\Lambda + 1)) \rfloor, \lfloor \text{dist}_{G_{T'}}(s,u)/(n \cdot (\Lambda + 1)) \rfloor$ are exactly the same as before scaling up the weights.

Algorithm 44 Approximate Single Source Shortest Path Tree and Potentials

procedure SSSP($G = (V, E, w), s \in V, \epsilon \in (0, 0.5)$)
  Output: $G_T = (V, T, w_T), \zeta \in \mathbb{R}^n$ ▷ Return a spanning tree and distance potentials of vertices.
  Initialize $\zeta \leftarrow \mathbf{0}_n$, let $G_T = (V, T, w_T)$ be an arbitrary spanning tree of G. Set $\epsilon' \leftarrow \epsilon/100$.
  Set $b_s \leftarrow -(n-1)$ and $\forall u \in V \setminus \{s\}, b_u \leftarrow 1$.
  while $b_s < 0$ **do**
    Compute a potential vector $\zeta' \in \mathbb{R}^n$ which is a $(1-\epsilon')$-approximate dual solution to the uncapacitated minimum cost flow problem on G with demand b.
    ▷ Use Theorem 6.4.28.
    Modify the obtained potentials $\zeta' \leftarrow \zeta' - \mathbf{1}_n \cdot \zeta'_s$ to make $\zeta'_s = 0$.
    Compute a spanning tree $G_{T'} = (V, T')$ of G such that $\sum_{u \in V \setminus \{s\}} b_u \cdot \mathrm{dist}_{G_{T'}}(s, u) \leq (1+\epsilon') \cdot \sum_{u \in V \setminus \{s\}} b_u \cdot \mathrm{dist}_G(s, u)$.
    ▷ Use Theorem 6.4.42.
    Compute a spanning tree $G_{T''} = (V, T'', w_{T''})$ of G such that $T'' \subseteq T \cup T', \forall e \in T'', w_{T''}(e) \leftarrow w(e)$ and $\forall u \in V, \mathrm{dist}_{G_{T''}}(s, u) \leq \min(\mathrm{dist}_{G_{T'}}(s, u), \mathrm{dist}_{G_T}(s, u))$.
    Update $G_T \leftarrow G_{T''}$, i.e., $T \leftarrow T'', w_T \leftarrow w_{T''}$.
    For each $u \in V \setminus \{s\}$, update $\zeta_u \leftarrow \max(\zeta_u, \zeta'_u)$.
    for $u \in V : b_u = 1$ **do**
      If $\mathrm{dist}_{G_{T'}}(s, u) \leq (1+\epsilon)\zeta'_u$, $b_u \leftarrow 0, b_s \leftarrow b_s + 1$.
    end for
  end while
  Return $G_T = (V, T, w_T)$ and ζ.
end procedure

We use the following procedure to compute T'':

1. Compute parent pointers $\mathrm{par} : V \setminus \{s\} \to V$ for G_T and parent pointers $\mathrm{par}' : V \setminus \{s\} \to V$ for $G_{T'}$, i.e., $\{\{u, \mathrm{par}(u)\} \mid u \in V \setminus \{s\}\} = T$, $\{\{u, \mathrm{par}'(u)\} \mid u \in V \setminus \{s\}\} = T'$.

2. For each vertex $u \in V \in \{s\}$, if $\mathrm{dist}_{G_T}(s, u) < \mathrm{dist}_{G_{T'}}(s, u)$, set $\mathrm{par}''(u) \leftarrow \mathrm{par}(u), T'' \leftarrow T'' \cup \{u, \mathrm{par}(u)\}$, $w_{T''}(u, \mathrm{par}''(u)) \leftarrow w_T(u, \mathrm{par}(u))$, otherwise set $\mathrm{par}''(u) \leftarrow \mathrm{par}'(u), T'' \leftarrow T'' \cup \{u, \mathrm{par}'(u)\}, w_{T''}(u, \mathrm{par}''(u)) \leftarrow w_{T'}(u, \mathrm{par}'(u))$.

For the first step, we can use $\mathrm{poly}(\log n)$ depth and $m\,\mathrm{poly}(\log n)$ work to find parent pointers [13]. In the second step, we can use the doubling algorithm to compute $\mathrm{dist}_{G_T}(s, u)$ and $\mathrm{dist}_{G_{T'}}(s, u)$ for each $u \in V$ in $n\,\mathrm{poly}(\log n)$ work and $\mathrm{poly}(\log n)$ depth. Thus, the whole algorithm takes $\mathrm{poly}(\log n)$ depth and $n \cdot \mathrm{poly}(\log n)$ work.

Now let us consider the correctness of the procedure. Firstly, we show that $G_{T''}$ is indeed a tree. Notice that for each vertex $u \in V \setminus \{s\}$, $\mathrm{par}''(u)$ is either $\mathrm{par}(u)$ or $\mathrm{par}'(u)$. If $\mathrm{par}''(u) =$

$\text{par}(u)$, then by the construction, we know that $\min(\text{dist}_{G_T}(s,u), \text{dist}_{G_{T'}}(s,u)) > \text{dist}_{G_T}(s, \text{par}''(u)) \geq \min(\text{dist}_{G_T}(s, \text{par}''(u)), \text{dist}_{G_{T'}}(s, \text{par}''(u)))$. Similarly, if $\text{par}''(u) = \text{par}'(u)$, then by the construction, we know that $\min(\text{dist}_{G_T}(s,u), \text{dist}_{G_{T'}}(s,u)) > \text{dist}_{G_{T'}}(s, \text{par}''(u)) \geq \min(\text{dist}_{G_T}(s, \text{par}''(u)), \text{dist}_{G_{T'}}(s, \text{par}''(u)))$. In either case, we have

$$\min\left(\text{dist}_{G_T}(s,u), \text{dist}_{G_{T'}}(s,u)\right) > \min\left(\text{dist}_{G_T}(s, \text{par}''(u)), \text{dist}_{G_{T'}}(s, \text{par}''(u))\right).$$

Thus, par'' does not create any cycle which implies that T'' is a tree.

We claim that $\forall u \in V$, $\text{dist}_{G_{T''}}(s,u) \leq \min(\text{dist}_{G_T}(s,u), \text{dist}_{G_{T'}}(s,u))$. The proof is by induction on $\min(\text{dist}_{G_T}(s,u), \text{dist}_{G_{T'}}(s,u))$. The base case is when $u = s$, the claim is obviously true. Suppose any $v \in V$ with $\min(\text{dist}_{G_T}(s,v), \text{dist}_{G_{T'}}(s,v)) < \min(\text{dist}_{G_T}(s,u), \text{dist}_{G_{T'}}(s,u))$ satisfies $\text{dist}_{G_{T''}}(s,v) \leq \min(\text{dist}_{G_T}(s,v), \text{dist}_{G_{T'}}(s,v))$. Consider vertex u. In the case $\text{par}''(u) = \text{par}(u)$, we have $\text{dist}_{G_T}(s,u) \leq \text{dist}_{G_{T'}}(s,u)$. We have $\text{dist}_{G''_T}(s,u) \leq \text{dist}_{G_{T''}}(s, \text{par}(u)) + w_T(u, \text{par}(u)) \leq \text{dist}_{G_T}(s, \text{par}(u)) + w_T(\text{par}(u), u) = \text{dist}_{G_T}(s,u) = \min(\text{dist}_{G_T}(s,u), \text{dist}_{G_{T'}}(s,u))$. In the case $\text{par}''(u) = \text{par}'(u)$, we have $\text{dist}_{G_{T'}}(s,u) \leq \text{dist}_{G_T}(s,u)$. We have $\text{dist}_{G''_T}(s,u) \leq \text{dist}_{G_{T''}}(s, \text{par}'(u)) + w_{T'}(u, \text{par}'(u)) \leq \text{dist}_{G_{T'}}(s, \text{par}'(u)) + w_{T'}(\text{par}'(u), u) = \text{dist}_{G_{T'}}(s,u) = \min(\text{dist}_{G_T}(s,u), \text{dist}_{G_{T'}}(s,u))$. Thus, the claim holds. □

Lemma 6.4.44 (Correctness of Algorithm 44)**.** *Given an $\epsilon \in (0, 0.5)$, a connected n-vertex m-edge undirected graph $G = (V, E, w)$ with $w : E \to \mathbb{Z}_{\geq 0}$ and a vertex s, if $\text{SSSP}(G, s, \epsilon)$ (Algorithm 44) terminates, the output $G_T = (V, T, w_T)$ and ζ satisfy:*

1. $\forall u \in V, \text{dist}_{G_T}(s,u) \leq (1+\epsilon) \cdot \text{dist}_G(s,u)$.
2. $\forall u \in V, \zeta_u \geq (1-\epsilon) \cdot \text{dist}_G(s,u)$ *and* $\forall \{u,v\} \in E, |\zeta_u - \zeta_v| \leq w(u,v)$.

Proof. Consider an arbitrary vertex u. If $\text{SSSP}(G, s, \epsilon)$ (Algorithm 44) terminates, it implies that $b_s = 0$ at the end of the algorithm and thus there is an iteration that the condition of line 15 is satisfied: $\text{dist}_{G_{T'}}(s,u) \leq (1+\epsilon)\zeta'_u$. Notice that ζ' is a dual solution obtained by line 6. After the modification of line 8, ζ' is still a dual solution. Thus, $\forall \{x,y\} \in E, |\zeta'_x - \zeta'_y| \leq w(x,y)$ which

implies that $\zeta'_u = \zeta'_u - \zeta'_s \leq \text{dist}_G(s,u)$. Since $\text{dist}_{G_{T'}}(s,u) \leq (1+\epsilon)\zeta'_u$, we have $\zeta'_u \geq (1-\epsilon)\,\text{dist}_G(s,u)$ and $\text{dist}_{G_{T'}}(s,u) \leq (1+\epsilon)\,\text{dist}_G(s,u)$.

According to line 11 and line 12, we have $\text{dist}_{G_T}(s,u) \leq \text{dist}_{G_{T'}}(s,u) \leq (1+\epsilon)\,\text{dist}_G(s,u)$ at the end of the algorithm. According to line 13, we have $\zeta_u \geq \zeta'_u \geq (1-\epsilon)\,\text{dist}_G(s,u)$.

Now we prove that $\forall\{x,y\} \in E, |\zeta_x - \zeta_y| \leq w(u,v)$ at the end of the algorithm. ζ is initialized as $\mathbf{0}_n$ which satisfies the property. The only place updates ζ is in line 13. Consider an arbitrary edge $\{x,y\} \in E$. Since ζ' is a feasible dual solution, we have both $|\zeta'_x - \zeta'_y| \leq w(x,y)$ and $|\zeta_x - \zeta_y| \leq w(x,y)$ before line 13. Notice that $|\max(\zeta_x,\zeta'_x) - \max(\zeta_y,\zeta'_y)| \leq \max(|\zeta_x - \zeta_y|, |\zeta'_x - \zeta'_y|) \leq w(x,y)$. Thus, ζ must hold the property at the end of the algorithm. □

Lemma 6.4.45 (Number of iterations of Algorithm 44)**.** *Given an $\epsilon \in (0, 0.5)$, a connected n-vertex m-edge undirected graph $G = (V,E,w)$ with $w : E \to \mathbb{Z}_{\geq 0}$ and a vertex s, the number of iterations of the while loop from line 5 to line 17 of* $\text{SSSP}(G,s,\epsilon)$ *(Algorithm 44) is at most $O(\log n\Lambda)$ where $\Lambda = \max_{e\in E} w(e)$.*

Proof. Consider one iteration. Let $\text{cost} = \sum_{u\in V\setminus\{s\}} b_s \cdot \text{dist}_G(s,u)$ at the beginning of the iteration. Let $\text{cost}' = \sum_{u\in V\setminus\{s\}} b_s \cdot \text{dist}_G(s,u)$ at the end of the iteration. Notice that $\text{cost} \leq \sum_{u\in V\setminus\{s\}} \text{dist}_G(s,u) \leq n^2\Lambda$. It suffices to show that $\text{cost} \geq \text{cost}' \cdot 2$.

Now we fix $b \in \mathbb{R}^n$ to be the demand vector at the beginning of the iteration. By line 6, line 8 and line 9, we have:

$$\sum_{u\in V\setminus\{s\}} b_u \cdot (\text{dist}_{G_{T'}}(s,u) - \text{dist}_G(s,u)) + \sum_{u\in V\setminus\{s\}} b_u \cdot (\text{dist}_G(s,u) - \zeta'_u) \leq 2\epsilon' \cdot \sum_{u\in V\setminus\{s\}} b_u \cdot \text{dist}_G(s,u).$$

It implies that

$$\sum_{u\in V\setminus\{s\}} b_u \cdot (\text{dist}_{G_{T'}}(s,u) - \zeta'_u) \leq \frac{\epsilon}{50} \cdot \sum_{u\in V\setminus\{s\}} b_u \cdot \text{dist}_G(s,u). \tag{6.20}$$

For a vertex $u \in V \setminus \{s\}$, if b_u is still 1 at the end of the iteration, we call it *bad*. If u is bad, then

by the condition of line 15, we have $\text{dist}_{G_T'}(s,u) > (1+\epsilon)\zeta'_u$ which implies that

$$\epsilon/2 \cdot \text{dist}_G(s,u) \leq \epsilon/2 \cdot \text{dist}_{G_T'}(s,u) \leq \text{dist}_{G_T'}(s,u) - \zeta'_u. \tag{6.21}$$

where the last step follows from $1/(1+\epsilon) \leq 1 - \epsilon/2$ for $\epsilon \in (0, 0.5)$. Thus, we have:

$$\begin{aligned}
\text{cost}' &= \sum_{u \in V \setminus \{s\}: u \text{ is bad}} b_u \cdot \text{dist}_G(s,u) \\
&\leq \frac{2}{\epsilon} \cdot \sum_{u \in V \setminus \{s\}: u \text{ is bad}} b_u \cdot (\text{dist}_{G_T'}(s,u) - \zeta'_u) \\
&\leq \frac{2}{\epsilon} \cdot \sum_{u \in V \setminus \{s\}} b_u \cdot (\text{dist}_{G_T'}(s,u) - \zeta'_u) \\
&\leq \frac{2}{\epsilon} \cdot \frac{\epsilon}{50} \cdot \sum_{u \in V \setminus \{s\}} b_u \cdot \text{dist}_G(s,u) \\
&= \frac{1}{25} \cdot \text{cost},
\end{aligned}$$

where the first step follows from the definition of cost′, the second step follows from Equation (6.21), the fourth step follows from Equation (6.20), and the last step follows from the definition of cost. □

Theorem 6.4.46 (Parallel single source shortest path tree and potentials)**.** *Given an $\epsilon \in (0, 0.5)$, a connected n-vertex m-edge undirected graph $G = (V, E, w)$ with $w : E \to \mathbb{Z}_{\geq 0}$ and a vertex s, there is a PRAM algorithm which takes $\epsilon^{-2}\,\text{poly}(\log(n\Lambda))$ parallel time using expected $\epsilon^{-3} m\,\text{poly}(\log(n\Lambda))$ work and with probability at least 0.99 outputs a spanning tree $G_T = (V, T, w_T)$ and a vector $\zeta \in \mathbb{R}^n$ satisfying following properties:*

1. $\forall u \in V, \text{dist}_{G_T}(s,u) \leq (1+\epsilon) \cdot \text{dist}_G(s,u)$.
2. $\forall u \in V, \zeta_u \geq (1-\epsilon) \cdot \text{dist}_G(s,u)$ *and* $\forall \{u,v\} \in E, |\zeta_u - \zeta_v| \leq w(u,v)$.

Proof. We invoke $\text{SSSP}(G, s, \epsilon)$ (Algorithm 44). By Lemma 6.4.45, $\text{SSSP}(G, s, \epsilon)$ terminates. If line 6 and line 9 succeed, the correctness follows from Lemma 6.4.44. According to Lemma 6.4.45,

the while loop from line 5 to line 17 takes $O(\log n)$ iterations. We invoke Theorem 6.4.28 for line 6 and invoke Theorem 6.4.42 for line 9. We can repeat line 6 and line 9 $\Theta(\log n)$ times to boost the success probability to $1-n^{-10}$. Since the while loop only has $O(\log n)$ iterations, we can take union bound over all iterations, the probability that we successfully compute ζ' and $G_{T'}$ in all iterations is at least 0.999.

Now consider the overall depth and work. We need $\text{poly}(\log n)$ depth and $m\,\text{poly}(\log n)$ work to compute an arbitrary spanning tree of G in line 3. According to Lemma 6.4.45, the while loop from line 5 to line 17 takes $O(\log n)$ iterations. In each iteration, we invoke algorithm described in Theorem 6.4.28 $\Theta(\log n)$ times for line 6. According to Theorem 6.4.28, the overall depth of line 6 is at most $\epsilon^{-2}\cdot\text{poly}(\log(n\Lambda))$ and the overall expected work is at most $\epsilon^{-2}\cdot m\cdot\text{poly}(\log(n\Lambda))$. In each iteration, we invoke algorithm described in Theorem 6.4.42 $\Theta(\log n)$ times for line 9. According to Theorem 6.4.42, the overall depth of line 9 is at most $\epsilon^{-2}\cdot\text{poly}(\log(n\Lambda))$ and the overall expected work is at most $\epsilon^{-3}\cdot m\cdot\text{poly}(\log(n\Lambda))$. According to lemma 6.4.43, the overall depth of line 11 is $\text{poly}(\log n)$ and the overall work is $n\,\text{poly}(\log n)$. As discussed in the proof of Lemma 6.4.43, we can compute $\text{dist}_{G_{T'}}(s,u)$ for all $u\in V$ simultaneously. Thus, the overall depth of line 15 is $\text{poly}(\log n)$ and the overall work is $n\,\text{poly}(\log n)$. To conclude, the overall depth of $\text{SSSP}(G,s,\epsilon)$ is $\epsilon^{-2}\,\text{poly}(\log(n\Lambda))$ and the overall expected work is $\epsilon^{-3}m\,\text{poly}(\log(n\Lambda))$. □

6.4.7 Massive parallel computing (MPC)

Although we present our parallel algorithms in the PRAM model, they can also be implemented in the Massive Parallel Computing (MPC) model [8, 1, 9, 2, 10] which is an abstract of massively parallel computing systems such as MapReduce [3], Hadoop [4], Dryad [101], Spark [102], and others. See Section 2.3 for a detailed description of the MPC model.

By applying the simulation methods [1, 9], our PRAM algorithm can be directly simulated in MPC. The obtained MPC algorithm has $\text{poly}(\log n)$ rounds and only needs $m\cdot\text{poly}(\log n)$ total space. Furthermore, it is also fully scalable, i.e., the memory size per machine can be allowed to be m^{δ} for any constant $\delta\in(0,1)$. To the best of our knowledge, this is the first MPC algorithm

which computes $(1+\epsilon)$-approximate shortest path using poly$(\log n)$ rounds and m poly$(\log n)$ total space when the memory of each machine is upper bounded by $n^{1-\Omega(1)}$. Previous work on shortest paths in the MPC model include [39] when the memory size per machine is $o(n)$, and simulations of shortest path algorithms from the Congested Clique model [40, 41, 42, 43, 44, 45] when the memory size per machine is $\Omega(n)$ [17].

Chapter 7: Hardness Results

In this chapter, we will see several hardness results for algorithms under parallel computing models.

7.1 Directed reachability vs. boolean matrix multiplication

In this section, we discuss the directed graph reachability problem which is a directed graph problem highly related to the undirected graph connectivity. In the all-pair directed graph reachability problem, we are given a directed graph $G = (V, E)$, the goal is to answer for every pair $(u, v) \in V \times V$ whether there is a directed path from u to v. There is a simple standard way to reduce Boolean Matrix Multiplication to all-pair directed graph reachability problem. In the Boolean Matrix Multiplication problem, we are given two boolean matrices $A, B \in \{0,1\}^{n \times n}$, the goal is to compute $C = A \cdot B$, where $\forall i, j \in [n], C_{i,j} = \bigvee_{k \in [n]} A_{i,k} \wedge B_{k,j}$. The reduction is as the following. We create $3n$ vertices $u_1, u_2, \cdots, u_n, v_1, v_2, \cdots, v_n, w_1, w_2, \cdots, w_n$. For every $i, j \in [n]$, if $A_{i,j} = 1$, then we add an edge from u_i to v_j, and if $B_{i,j} = 1$, then we add an edge from v_i to w_j. Thus, $C_{i,j} = 1$ is equivalent to there is a path from u_i to w_j. Thus, if we can solve all-pair directed graph reachability problem in $O(T)$ sequential time, then we can solve Boolean Matrix Multiplication in $O(T)$ time. For the current status of sequential running time of Boolean Matrix Multiplication problem, we refer readers to [103] and the references therein.

Now, consider the multi-query directed graph reachability problem. In this problem, we are given a directed graph $G = (V, E)$ together with $|V| + |E|$ queries where each query queries the reachability from vertex u to vertex v. The goal is to answer all these queries. A similar problem in the undirected graph is called multi-query undirected graph connectivity problem. In this problem, we are given an undirected graph $G = (V, E)$ together with $|V| + |E|$ queries where each query

queries the connectivity between vertex u and vertex v.

According to Theorem 4.4.4 and Lemma 2.3.6, there is a polynomial local running time fully scalable $\sim \log D$ parallel time $(0,\delta)-$MPC algorithm for multi-query undirected graph connectivity problem. Here polynomial local running time means that there is a constant $c > 0$ (independent from δ) such that every machine in one round can only have $O((n^{\delta})^{c})$ local computation.

For multi-query directed graph reachability problem, we show that if there is a polynomial local running time fully scalable (γ,δ) – MPC algorithm which can solve multi-query reachability problem in $O(n^{\alpha})$ parallel time, then we can solve all-pair directed graph reachability problem in $O(n^2 \cdot n^{2\gamma+\alpha+\epsilon})$ sequential running time for any arbitrarily small constant $\epsilon > 0$. Especially, if the algorithm is in $(0,\delta)$ – MPC model, and the parallel time is $n^{o(1)}$, then we will have an $O(n^{2+\epsilon+o(1)})$ sequential running time algorithm for Boolean Matrix Multiplication which implies a break through in this field.

Suppose we have a such MPC algorithm. Let the input size be $\Theta(m)$, i.e. the number of edges is $\Theta(m)$, and the number of queries is also $\Theta(m)$. Then the total space is $\Theta(m^{1+\gamma})$. Let $\delta = \epsilon/(c-2)$. Then the number of machines is $\Theta(m^{1+\gamma-\delta})$. Now we just simulate this (γ,δ) – MPC algorithm sequentially, the total running time is $O(m^{1+\gamma-\delta} \cdot m^{c\delta} \cdot n^{\alpha}) = O(m \cdot n^{2\gamma+\epsilon+\alpha})$. To answer reachability for all pairs, we need total $O(n^2 \cdot m \cdot n^{2\gamma+\epsilon+\alpha}/m) = O(n^2 \cdot n^{2\gamma+\alpha+\epsilon})$ time. Therefore, we can use this algorithm to solve Boolean Matrix Multiplication in $O(n^2 \cdot n^{2\gamma+\alpha+\epsilon})$ time.

Theorem 7.1.1. *If there is a polynomial local running time fully scalable* (γ,δ) – MPC *algorithm which can answer* $|V| + |E|$ *pairs of reachability queries simultaneously for any directed graph* $G = (V,E)$ *in* $O(|V|^{\alpha})$ *parallel time, then there is a sequential algorithm which can compute the multiplication of two* $n \times n$ *boolean matrices in* $O(n^2 \cdot n^{2\gamma+\alpha+\epsilon})$ *time, where* $\epsilon > 0$ *is a constant which can be arbitrarily small.*

Proof. See above discussions. □

7.2 Discussion on a previous conjectured fast algorithm

In this section, we discuss the hard example for the algorithm described by [76]. In [76], they conjectured that their Hash-to-Min connectivity algorithm can finish in $O(\log D)$ rounds. The description of their algorithm is as the following:

1. The input graph is $G = (V, E)$.
2. For each vertex $v \in V$, initialize a set $S_v^{(0)} = v$.
3. in round i:
   (a) Each vertex v find $u \in S_v^{(i-1)}$ which has the minimum label, i.e. $u = \min_{x \in S_v^{(i-1)}} x$.
   (b) v sends u the all the vertices in $S_v^{(i-1)}$.
   (c) v sends every $x \in S_v^{((i-1))} \setminus \{u\}$ the vertex u.
   (d) Let $S_v^{(i)}$ be $\{v\}$ union the set of all the vertices received.
   (e) If for all v, $S_v^{(i)}$ is the same as $S_v^{(i-1)}$, then finish the procedure.

The above procedure can be seen as the modification of the graph: in each round, all the vertices together create a new graph. For each vertex v, let u be the neighbor of v with the minimum label, and if x is a neighbor of v, then add an edge between x and u in the new graph. So in each round, each vertex just communicates with its neighbors to update the new minimum neighbor it learned. At the end of the algorithm, it is obvious that the minimum vertex in each component will have all the other vertices in that component, and for each non minimum vertex, it will have the minimum vertex in the same component.

A hard example for this algorithm is shown by Figure 7.1. The example is a thin and tall grid graph with a vertex connected to all the vertices in the first column. The total number of vertices is n. The grid graph has $D = \frac{1}{2}\log n$ columns and n/D rows. We index each column from left to right by 1 to D. We index each row from top to down by 1 to n/D. The single large degree vertex has label 0. The i^{th} row has the vertices with label $(i-1) \cdot D + 1$ to $i \cdot D$ from the first

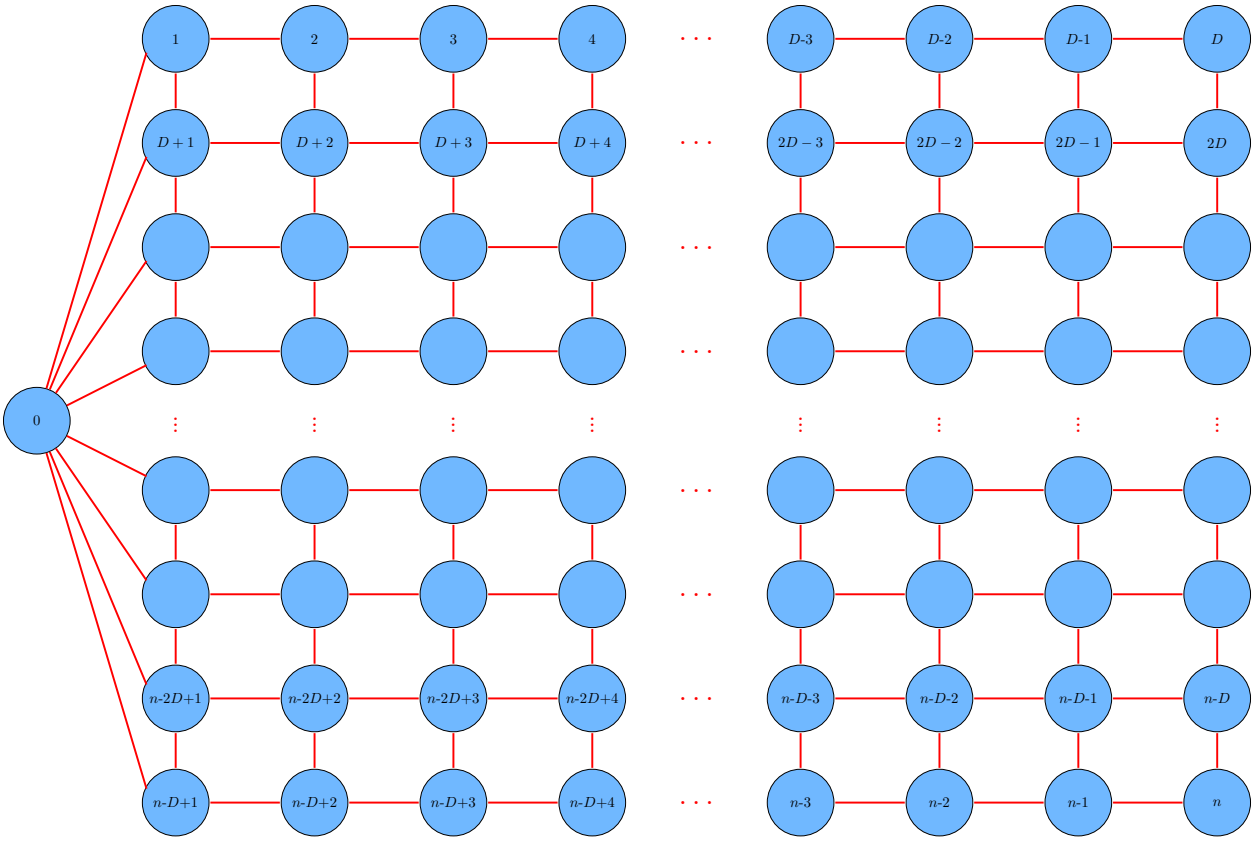

Figure 7.1: A hard example for [76]. For each $i \in \{2, 3, \cdots, n/D - 1\}$ and $j \in \{1, 2, \cdots, D - 1\}$, node $(i - 1) \cdot D + j$ has degree 4. For node D and n, they have degree 2. Node 0 has degree D. All the other nodes have degree 3.

column to the D^{th} column. We claim that if vertex v is the i^{th} row and j^{th} column, then before round k for $2^k < i, k < j$, the neighbors of v will only in column $j - 1$, column j and column $j + 1$. Furthermore, the minimum neighbor of v in column $j - 1$ will be $v - (2^{k-1} - 1) \cdot D - 1$. The minimum neighbor of v in column j will be $v - 2^{k-1} \cdot D$. The minimum neighbor of v in column $j + 1$ will be $v - D \cdot (2^{k-1} - 1) + 1$. This claim is true when $k = 1$. Then by induction, we can prove the claim. Thus, it will take at least $\Theta(D)$ rounds to finish the procedure where $D = \Theta(\log n)$.

If we randomly label the vertices at the beginning, then consider the case we copy that hard structure at least n^{n+2} times, then with high probability, there is a component which has the labels with the order as the same as described above. In this case, the procedure needs $\Omega(\log \log N)$ rounds, where $N = n^{n+3}$ is the total number of the vertices.

Also notice that, even we give more total space to this algorithm, this algorithm will not preform

better. In our connectivity algorithm, if we have $\Omega(n^{1+\epsilon})$ total space for some arbitrary constant $\epsilon > 0$, then our parallel running time is $O(\log D)$.

7.3 Hardness of biconnectivity in MPC

There is a conjectured hardness result which is widely used in the MPC literature [1, 2, 47, 48, 49].

Conjecture 7.3.1 (One cycle vs. two cycles)**.** *For any $\gamma \geq 0$ and any constant $\delta \in (0, 1)$, distinguishing the following two graph instances in the (γ, δ)-MPC model requires $\Omega(\log n)$ parallel time:*

1. *a single cycle contains n vertices,*
2. *two disjoint cycles, each contains $n/2$ vertices.*

Under the above conjecture, we show that $\Omega(\log \text{bi-diam}(G))$ parallel time is necessary to compute the biconnected components of G. This claim is true even for the constant diameter graph G, i.e., $\text{diam}(G) = O(1)$.

Theorem 7.3.2 (Hardness of biconnectivity in MPC)**.** *For any $\gamma \geq 0$ and any constant $\delta \in (0, 1)$, unless the one cycle vs. two cycles conjecture (Conjecture 7.3.1) is false, any (γ, δ)-MPC algorithm requires $\Omega(\log \text{bi-diam}(G))$ parallel time for testing whether a graph G with a constant diameter is biconnected.*

Proof. For $\gamma \geq 0$ and an arbitrary constant $\delta \in (0, 1)$, suppose there is a (γ, δ)-MPC algorithm $\mathcal{A}$ which can determine whether an arbitrary constant diameter graph G is biconnected in $o(\log \text{bi-diam}(G))$ parallel time. Then we give a (γ, δ)-MPC algorithm for solving one cycle vs. two cycles problem as the following:

1. For a one cycle vs. two cycles instance n-vertex graph $G' = (V', E')$, construct a new graph $G = (V, E)$: $V = V' \cup \{v^*\}, E = E' \cup \{(v, v^*) \mid v \in V'\}$.

2. Run $\mathcal{A}$ on G. If G is not biconnected, G' contains two cycles. Otherwise G' is a single cycle.

It is easy to see that the diameter of G is 2. If G' is a single cycle, then G is biconnected and bi-diam$(G) = \Theta(n)$. If G' contains two cycles, then G contains two biconnected components and bi-diam$(G) = \Theta(n)$.

The first step of the above algorithm takes $O(1)$ parallel time and only requires linear total space. The graph G has $n + 1$ vertices and $2n$ edges. Thus, the above algorithm is also a (γ, δ)-MPC algorithm. The parallel time of the above algorithm is the same as the time needed for running $\mathcal{A}$ on G which is $o(\log \text{bi-diam}(G)) = o(\log n)$. Thus the existence of the algorithm $\mathcal{A}$ implies that the one cycle vs. two cycles conjecture (Conjecture 7.3.1) is false. □

7.4 The necessity of 2 types of edges in the subemulator

We show that both types of edges constructed by line 5 and line 6 in Algorithm 32 are necessary for the construction of subemulator. If we only contain the edges constructed by line 5 and miss the edges constructed by line 6, Figure 7.2 gives an example that the constructed graph can not be a subemulator.

If we only contain the edges constructed by line 6 and miss the edges constructed by line 5, Figure 7.3 gives an example that the constructed graph can not be a subemulator.

7.5 Connectivity in CREW PRAM

In this section we show that any deterministic CREW PRAM algorithm which solves connectivity must take $\Omega(\log n)$ parallel time even when the input graph has diameter at most 2. We also show that any randomized CREW PRAM algorithm which solves connectivity using linear number of processors with probability at least $2/3$ must take $\Omega(\log \log n)$ parallel time even when the input graph has diameter at most 2. The proof is by reduction from computing OR on n-bits.

Theorem 7.5.1 ([104])**.** *Any deterministic CREW PRAM algorithm which computes OR on n-bits must take* $\Omega(\log n)$ *parallel time even when the number of processors and the number of shared*

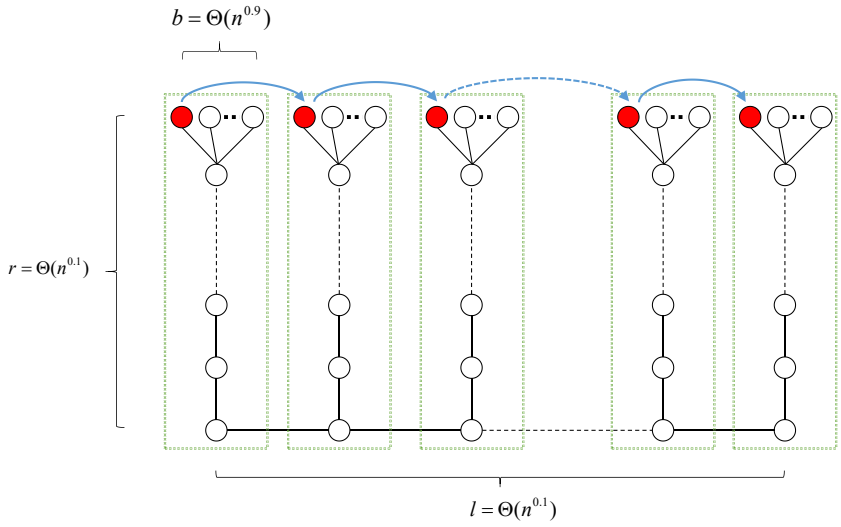

Figure 7.2: The graph is unweighted and is a tree constructed by following steps. We first create a path with length $l = \Theta(n^{0.1})$. For each vertex on the path, we create a brunch starting with a path with length $r = \Theta(n^{0.1})$ and ending with a star with $b = \Theta(n^{0.9})$ vertices. If we sample each vertex (solid red vertex) to be in the subemulator with probability $\log(n)/b$, with high probability, sampled vertices can only appear in stars and each brunch must have at least one sampled vertex. We condition on this event. It is clear that each vertex has at least one $(b + r)$-closest neighbor which is a sampled vertex, and that sampled vertex must be in the same dashed green box. If we only contain the edges constructed by line 5 of Algorithm 32, the result graph must be a length-l path (represented by blue arcs) where each edge corresponds to an edge crossing two dashed green box above and has weight $2r + 1$. Thus the diameter of the result graph is $l(2r + 1) = \Theta(n^{0.2})$. However, the diameter of the original graph is $2r + l = \Theta(n^{0.1})$ which implies that the result graph is not a good subemulator.

memory cells are unlimited.

Theorem 7.5.2 ([105])**.** *Any randomized CREW PRAM algorithm which computes OR on n-bits using n processors with successful probability at least* $2/3$ *must take* $\Omega(\log \log n)$ *parallel time.*

We show that computing OR on n-bits can be reduced to determine whether two fixed vertices in the input graph with diameter at most 2 is connected or not. The reduction is shown as the following: For each bit we create a vertex x_i. We also create two vertices s and t. We create an edge between every x_i and s. We create an edge between x_i and t only if $x_i = 1$. Notice that these steps can be done deterministically in CREW PRAM using $O(1)$ parallel time and $O(n)$ processors. Notice that the outcome of OR on $x_1, x_2, \cdots, x_n$ is 1 if and only if s and t are in the same connected component. Furthermore, the constructed graph has diameter at most 2. Thus, we can conclude

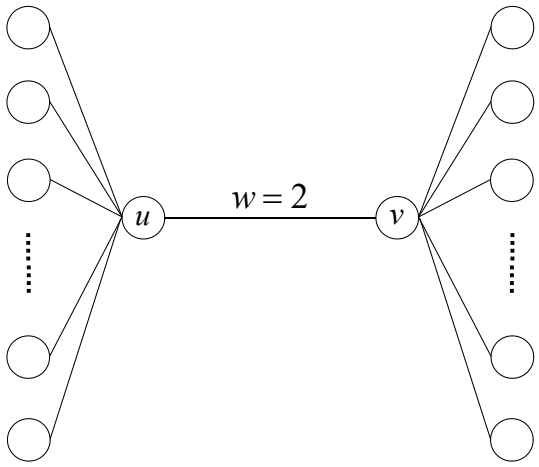

Figure 7.3: The graph contains two stars connecting by an edge with weight 2. Each star has $n/2$ vertices. One star has center u and another has center v. Except the edge between u, v, all other edges have weights 1. For $b < n/2$, neither v is a b-closest neighbor of any vertex in the star with center u nor u is a b-closest neighbor of any vertex in the star with center v. Thus, if we only contain the edges constructed by line 6 of Algorithm 32, the result graph is disconnected which cannot be a subemulator.

the following two theorem.

Theorem 7.5.3. *Any deterministic CREW PRAM algorithm which solves connectivity for an n-vertex graph with diameter at most 2 needs $\Omega(\log n)$ parallel time even when the number of processors and the number of shared memory cells are unlimited.*

Theorem 7.5.4. *Any randomized CREW PRAM algorithm which uses $O(n)$ processors to solve connectivity for an n-vertex graph with diameter 2 with successful probability at least $2/3$ needs $\Omega(\log \log n)$ parallel time.*

References

[1] H. Karloff, S. Suri, and S. Vassilvitskii, "A model of computation for mapreduce", in *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, Society for Industrial and Applied Mathematics, 2010, pp. 938–948.

[2] P. Beame, P. Koutris, and D. Suciu, "Communication steps for parallel query processing", in *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, ACM, 2013, pp. 273–284.

[3] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters", *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks", *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 59–72, 2007.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets.", *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[7] A. Gibbons and W. Rytter, *Efficient parallel algorithms*. Cambridge University Press, 1989.

[8] J. Feldman, S. Muthukrishnan, A. Sidiropoulos, C. Stein, and Z. Svitkina, "On distributing symmetric streaming computations", *ACM Transactions on Algorithms*, vol. 6, no. 4, 2010, Previously in SODA'08.

[9] M. T. Goodrich, N. Sitchinava, and Q. Zhang, "Sorting, searching, and simulation in the mapreduce framework", in *ISAAC*, Springer, vol. 7074, 2011, pp. 374–383.

[10] A. Andoni, A. Nikolov, K. Onak, and G. Yaroslavtsev, "Parallel algorithms for geometric graph problems", in *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, 2014, pp. 574–583.

[11] L. G. Valiant, "A bridging model for parallel computation", *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[12] P. Beame and J. Håstad, "Optimal bounds for decision problems on the CRCW PRAM", *J. ACM*, vol. 36, no. 3, pp. 643–670, 1989.

[13] Y. Shiloach and U. Vishkin, "An o (logn) parallel connectivity algorithm", *Journal of Algorithms*, vol. 3, no. 1, pp. 57–67, 1982.

[14] S. Halperin and U. Zwick, "An optimal randomised logarithmic time connectivity algorithm for the erew pram", *Journal of Computer and System Sciences*, vol. 53, no. 3, pp. 395–416, 1996.

[15] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii, "Filtering: A method for solving graph problems in MapReduce", in *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, ACM, 2011, pp. 85–94.

[16] T. Jurdziński and K. Nowicki, "Mst in o (1) rounds of congested clique", in *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, 2018, pp. 2620–2632.

[17] S. Behnezhad, M. Derakhshan, and M. Hajiaghayi, "Brief announcement: Semi-mapreduce meets congested clique. corr, abs/1802.10297, 2018", *arXiv preprint arXiv:1802.10297*, 2018.

[18] S. Behnezhad, L. Dhulipala, H. Esfandiari, J. Lacki, and V. Mirrokni, "Near-optimal massively parallel graph connectivity", in *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE, 2019, pp. 1615–1636.

[19] S. C. Liu, R. E. Tarjan, and P. Zhong, "Connected components on a pram in log diameter time", in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2020, pp. 359–369.

[20] S. Pettie and V. Ramachandran, "A randomized time-work optimal parallel algorithm for finding a minimum spanning forest", *SIAM Journal on Computing*, vol. 31, no. 6, pp. 1879–1895, 2002.

[21] R. E. Tarjan and U. Vishkin, "Finding biconnected componemts and computing tree functions in logarithmic parallel time", in *25th Annual Symposium onFoundations of Computer Science, 1984.*, IEEE, 1984, pp. 12–20.

[22] G. Ausiello, D. Firmani, L. Laura, and E. Paracone, "Large-scale graph biconnectivity in mapreduce", *Department of Computer and System Sciences Antonio Ruberti Technical Reports*, vol. 4, no. 4, 2012.

[23] R. Diestel, *Graph theory*. Springer Publishing Company, Incorporated, 2018.

[24] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms", *Journal of the ACM (JACM)*, vol. 34, no. 3, pp. 596–615, 1987.

[25] M. Thorup, "Undirected single-source shortest paths with positive integer weights in linear time", *Journal of the ACM (JACM)*, vol. 46, no. 3, pp. 362–394, 1999.

[26] T. D. Hansen, H. Kaplan, R. E. Tarjan, and U. Zwick, "Hollow heaps", in *International Colloquium on Automata, Languages, and Programming*, Springer, 2015, pp. 689–700.

[27] T. H. Spencer, "Time-work tradeoffs for parallel algorithms", *Journal of the ACM (JACM)*, vol. 44, no. 5, pp. 742–778, 1997.

[28] P. N. Klein and S. Subramanian, "A randomized parallel algorithm for single-source shortest paths", *Journal of Algorithms*, vol. 25, no. 2, pp. 205–220, 1997.

[29] E. Cohen, "Using selective path-doubling for parallel shortest-path computations", *Journal of Algorithms*, vol. 22, no. 1, pp. 30–56, 1997.

[30] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis, "A parallel priority queue with constant time operations", *Journal of Parallel and Distributed Computing*, vol. 49, no. 1, pp. 4–21, 1998.

[31] H. Shi and T. H. Spencer, "Time–work tradeoffs of the single-source shortest paths problem", *Journal of algorithms*, vol. 30, no. 1, pp. 19–32, 1999.

[32] G. E. Blelloch, Y. Gu, Y. Sun, and K. Tangwongsan, "Parallel shortest paths using radius stepping", in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, ACM, 2016, pp. 443–454.

[33] S. Forster and D. Nanongkai, "A faster distributed single-source shortest paths algorithm", in *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE, 2018, pp. 686–697.

[34] E. Cohen, "Polylog-time and near-linear work approximation scheme for undirected shortest paths", in *Proceedings of the 26th Annual ACM SIGACT Symposium on Theory of Computing*, vol. 26, 1994, pp. 16–26.

[35] P. N. Klein and S. Sairam, "A parallel randomized approximation scheme for shortest paths", in *Proceedings of the 24th Annual ACM SIGACT Symposium on Theory of Computing*, vol. 92, 1992, pp. 750–758.

[36] E. Cohen, "Polylog-time and near-linear work approximation scheme for undirected shortest paths", *Journal of the ACM (JACM)*, vol. 47, no. 1, pp. 132–166, 2000.

[37] G. L. Miller, R. Peng, A. Vladu, and S. C. Xu, "Improved parallel algorithms for spanners and hopsets", in *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, ACM, 2015, pp. 192–201.

[38] M. Elkin and O. Neiman, "Hopsets with constant hopbound, and applications to approximate shortest paths", in *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE, 2016, pp. 128–137.

[39] M. Dinitz and Y. Nazari, "Brief announcement: Massively parallel approximate distance sketches", in *33rd International Symposium on Distributed Computing (DISC 2019)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[40] Z. Lotker, B. Patt-Shamir, E. Pavlov, and D. Peleg, "Minimum-weight spanning tree construction in o (log log n) communication rounds", *SIAM Journal on Computing*, vol. 35, no. 1, pp. 120–131, 2005.

[41] H. Klauck, D. Nanongkai, G. Pandurangan, and P. Robinson, "Distributed computation of large-scale graph problems", in *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics, 2015, pp. 391–410.

[42] M. Henzinger, S. Krinninger, and D. Nanongkai, "An almost-tight distributed algorithm for computing single-source shortest paths. 2016", STOC, 2016.

[43] R. Becker, A. Karrenbauer, S. Krinninger, and C. Lenzen, "Near-optimal approximate shortest paths and transshipment in distributed and streaming models", in *31st International Symposium on Distributed Computing (DISC 2017)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[44] K. Censor-Hillel, M. Dory, J. H. Korhonen, and D. Leitersdorf, "Fast approximate shortest paths in the congested clique", in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, ser. PODC '19, Toronto ON, Canada: ACM, 2019, pp. 74–83, ISBN: 978-1-4503-6217-7.

[45] M. Henzinger, S. Krinninger, and D. Nanongkai, "A deterministic almost-tight distributed algorithm for approximating single-source shortest paths", *SIAM Journal on Computing*, no. 0, STOC16–98, 2019.

[46] J. Sherman, "Generalized preconditioning and undirected minimum-cost flow", in *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, 2017, pp. 772–780.

[47] R. Kiveris, S. Lattanzi, V. Mirrokni, V. Rastogi, and S. Vassilvitskii, "Connected components in mapreduce and beyond", in *Proceedings of the ACM Symposium on Cloud Computing*, ACM, 2014, pp. 1–13.

[48] T. Roughgarden, S. Vassilvitskii, and J. R. Wang, "Shuffles and circuits:(on lower bounds for modern parallel computation)", in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, ACM, 2016, pp. 1–12.

[49] G. Yaroslavtsev and A. Vadapalli, "Massively parallel algorithms and hardness for single-linkage clustering under lp distances", in *International Conference on Machine Learning*, 2018, pp. 5596–5605.

[50] R. E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm", *SIAM Journal on Computing*, vol. 14, no. 4, pp. 862–874, 1985.

[51] O. O'Malley, "Terabyte sort on apache hadoop", *Yahoo Tech. Rep*, 2008.

[52] M. Thorup and U. Zwick, "Spanners and emulators with sublinear distance errors", in *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, Society for Industrial and Applied Mathematics, 2006, pp. 802–809.

[53] S.-E. Huang and S. Pettie, "Thorup–zwick emulators are universally optimal hopsets", *Information Processing Letters*, vol. 142, pp. 9–13, 2019.

[54] A. Abboud, G. Bodwin, and S. Pettie, "A hierarchy of lower bounds for sublinear additive spanners", *SIAM Journal on Computing*, vol. 47, no. 6, pp. 2203–2236, 2018.

[55] J. Bourgain, "On lipschitz embedding of finite metric spaces in hilbert space", *Israel Journal of Mathematics*, vol. 52, no. 1-2, pp. 46–52, 1985.

[56] J. Fakcharoenphol, S. Rao, and K. Talwar, "A tight bound on approximating arbitrary metrics by tree metrics", *Journal of Computer and System Sciences*, vol. 69, no. 3, pp. 485–497, 2004.

[57] S. Friedrichs and C. Lenzen, "Parallel metric tree embedding based on an algebraic view on moore-bellman-ford", *Journal of the ACM (JACM)*, vol. 65, no. 6, p. 43, 2018.

[58] G. L. Miller, R. Peng, and S. C. Xu, "Parallel graph decompositions using random shifts", in *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, ACM, 2013, pp. 196–203.

[59] S. I. Daitch and D. A. Spielman, "Faster approximate lossy generalized flow via interior point algorithms", in *Proceedings of the fortieth annual ACM symposium on Theory of computing*, ACM, 2008, pp. 451–460.

[60] P. Christiano, J. A. Kelner, A. Madry, D. A. Spielman, and S.-H. Teng, "Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs", in *Proceedings of the forty-third annual ACM symposium on Theory of computing*, ACM, 2011, pp. 273–282.

[61] J. Sherman, "Nearly maximum flows in nearly linear time", in *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, IEEE, 2013, pp. 263–269.

[62] A. Madry, "Navigating central path with electrical flows: From flows to matchings, and back", in *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, IEEE, 2013, pp. 253–262.

[63] J. A. Kelner, Y. T. Lee, L. Orecchia, and A. Sidford, "An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations", in *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, SIAM, 2014, pp. 217–226.

[64] Y. T. Lee and A. Sidford, "Path finding methods for linear programming: Solving linear programs in Õ (sqrt(rank)) iterations and faster algorithms for maximum flow", in *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, IEEE, 2014, pp. 424–433.

[65] M. B. Cohen, A. Mądry, P. Sankowski, and A. Vladu, "Negative-weight shortest paths and unit capacity minimum cost flow in Õ (m 10/7 log w) time*", in *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, 2017, pp. 752–771.

[66] J. Sherman, "Area-convexity, l_∞ regularization, and undirected multicommodity flow", in *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, ACM, 2017, pp. 452–460.

[67] A. B. Khesin, A. Nikolov, and D. Paramonov, "Preconditioning for the geometric transportation problem", *arXiv preprint arXiv:1902.08384*, 2019.

[68] A. Andoni, Z. Song, C. Stein, Z. Wang, and P. Zhong, "Parallel graph connectivity in log diameter rounds", in *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE, 2018, pp. 674–685.

[69] A. Andoni, C. Stein, and P. Zhong, "Log diameter rounds algorithms for 2-vertex and 2-edge connectivity", in *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[70] ——, "Parallel approximate undirected shortest paths via low hop emulators", in *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, 2020, pp. 322–335.

[71] F. E. Fich, P. Ragde, and A. Wigderson, "Relations between concurrent-write models of parallel computation", *SIAM Journal on Computing*, vol. 17, no. 3, pp. 606–627, 1988.

[72] U. Vishkin, "Implementation of simultaneous memory address access in models that forbid it", *Journal of algorithms*, vol. 4, no. 1, pp. 45–50, 1983.

[73] F. E. Fich, P. Ragde, and A. Wigderson, "Relations between concurrent-write models of parallel computation", *SIAM Journal on Computing*, vol. 17, no. 3, pp. 606–627, 1988.

[74] R. M. Karp and V. Ramachandran, “A survey of parallel algorithms for shared-memory machines”, 1989.

[75] M. T. Goodrich, “Communication-efficient parallel sorting”, *SIAM Journal on Computing*, vol. 29, no. 2, pp. 416–432, 1999.

[76] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma, “Finding connected components in map-reduce in logarithmic rounds”, in *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, IEEE, 2013, pp. 50–61.

[77] H. Gazit, “An optimal randomized parallel algorithm for finding connected components in a graph”, *SIAM J. Comput.*, vol. 20, no. 6, pp. 1046–1067, 1991.

[78] S. Halperin and U. Zwick, “An optimal randomised logarithmic time connectivity algorithm for the EREW PRAM”, *J. Comput. Syst. Sci.*, vol. 53, no. 3, pp. 395–416, 1996.

[79] ——, “Optimal randomized erew pram algorithms for finding spanning forests”, *Journal of Algorithms*, vol. 39, no. 1, pp. 1–46, 2001.

[80] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, “Computing connected components on parallel computers”, *Commun. ACM*, vol. 22, no. 8, pp. 461–464, 1979.

[81] G. L. Miller and J. H. Reif, “Parallel tree contraction and its application”, in *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*, 1985, pp. 478–489.

[82] R. E. Tarjan and J. van Leeuwen, “Worst-case analysis of set union algorithms”, *J. ACM*, vol. 31, no. 2, pp. 245–281, 1984.

[83] S. C. Liu and R. 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*, 2019, 3:1–3:20.

[84] S. G. Akl, *Design and analysis of parallel algorithms*. Prentice Hall, 1989.

[85] J. H. Reif, “Optimal parallel algorithms for graph connectivity.”, HARVARD UNIV CAMBRIDGE MA AIKEN COMPUTATION LAB, Tech. Rep., 1984.

[86] F. E. Fich, F. Meyer auf der Heide, P. Ragde, and A. 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*, 1985, pp. 48–58.

[87] H. Whitney, “Non-separable and planar graphs”, *Transactions of the American Mathematical Society*, vol. 34, no. 2, pp. 339–362, 1932.

[88] V. Ramachandran, *Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity*. Citeseer, 1992.

[89] M. Thorup and U. Zwick, "Approximate distance oracles", *Journal of the ACM (JACM)*, vol. 52, no. 1, pp. 1–24, 2005.

[90] A. Bernstein, "Fully dynamic (2+eps) approximate all-pairs shortest paths with fast query and close to linear update time", in *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, IEEE, 2009, pp. 693–702.

[91] M. Henzinger, S. Krinninger, and D. Nanongkai, "Decremental single-source shortest paths on undirected graphs in near-linear total update time", in *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, 2014.

[92] A. Moitra, "Approximation algorithms for multicommodity-type problems with guarantees independent of the graph size", in *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, IEEE, 2009, pp. 3–12.

[93] F. T. Leighton and A. Moitra, "Extensions and limits to vertex sparsification", in *Proceedings of the forty-second ACM symposium on Theory of computing*, ACM, 2010, pp. 47–56.

[94] R. Krauthgamer, H. Nguyen, and T. Zondiner, "Preserving terminal distances using minors", *SIAM Journal on Discrete Mathematics*, vol. 28, no. 1, pp. 127–141, 2014.

[95] S. Arora, E. Hazan, and S. Kale, "The multiplicative weights update method: A meta-algorithm and applications", *Theory of Computing*, vol. 8, no. 1, pp. 121–164, 2012.

[96] P. Indyk and N. Thaper, "Fast image retrieval via embeddings", in *Workshop on Statistical and Computational Theories of Vision (at ICCV)*, 2003.

[97] A. Bačkurs and P. Indyk, "Better embeddings for planar earth-mover distance over sparse sets", in *Proceedings of the thirtieth annual symposium on Computational geometry*, ACM, 2014, p. 280.

[98] S. Chung and A. Condon, "Parallel implementation of bouvka's minimum spanning tree algorithm", in *Proceedings of International Conference on Parallel Processing*, IEEE, 1996, pp. 302–308.

[99] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[100] J. Li, "Faster parallel algorithm for approximate shortest path", in *Proceedings of the ACM SIGACT Symposium on Theory of Computing*, First appeared as arXiv:1911.01626, 2020.

[101] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks", in *ACM SIGOPS operating systems review*, ACM, vol. 41, 2007, pp. 59–72.

[102] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets.", *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[103] F. Le Gall, "Powers of tensors and fast matrix multiplication", in *Proceedings of the 39th international symposium on symbolic and algebraic computation*, ACM, 2014, pp. 296–303.

[104] S. Cook, C. Dwork, and R. Reischuk, "Upper and lower time bounds for parallel random access machines without simultaneous writes", *SIAM Journal on Computing*, vol. 15, no. 1, pp. 87–97, 1986.

[105] P. D. MacKenzie, "Lower bounds for randomized exclusive write prams", *Theory of Computing Systems*, vol. 30, no. 6, pp. 599–626, 1997.