Log Diameter Rounds Algorithms for 2-Vertex and 2-Edge Connectivity

Alexandr Andoni
Columbia University, New York City, NY, USA
andoni@cs.columbia.edu

Clifford Stein
Columbia University, New York City, NY, USA
cliff@cs.columbia.edu

Peilin Zhong
Columbia University, New York City, NY, USA
pz2225@columbia.edu

Abstract

Many modern parallel systems, such as MapReduce, Hadoop and Spark, can be modeled well by the MPC model. 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 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. This model is stronger than the classical PRAM model, and it is an intriguing question to design algorithms whose running time is smaller than in the PRAM model.

In this paper, we study two fundamental problems, 2-edge connectivity and 2-vertex connectivity (biconnectivity). PRAM algorithms which run in $O(\log n)$ time have been known for many years. We give algorithms using roughly log diameter rounds in the MPC model. Our main results are, for an n-vertex, m-edge graph of diameter D and bi-diameter D', 1) a $O(\log D \log\log_{m/n} n)$ parallel time 2-edge connectivity algorithm, 2) a $O(\log D \log^2 \log_{m/n} n + \log D' \log\log_{m/n} n)$ parallel time biconnectivity algorithm, where the bi-diameter D' is the largest cycle length over all the vertex pairs in the same biconnected component. Our results are fully scalable, meaning that the memory per processor can be $O(n^\delta)$ for arbitrary constant $\delta > 0$, and the total memory used is linear in the problem size. Our 2-edge connectivity algorithm achieves the same parallel time as the connectivity algorithm of [4]. We also show an $\Omega(\log D')$ conditional lower bound for the biconnectivity problem.

2012 ACM Subject Classification Theory of computation → MapReduce algorithms; Mathematics of computing → Paths and connectivity problems

Keywords and phrases parallel algorithms, biconnectivity, 2-edge connectivity, the MPC model

Digital Object Identifier 10.4230/LIPIcs.ICALP.2019.14

Category Track A: Algorithms, Complexity and Games

Related Version A full version of the paper is available at `https://arxiv.org/abs/1905.00850`.

Funding *Alexandr Andoni*: Research partly supported by NSF Grants (CCF-1617955 and CCF-1740833), Simons Foundation (#491119) and Google Research Award.
Clifford Stein: Research partly supported by NSF Grants CCF-1714818 and CCF-1822809.
Peilin Zhong: Research partly supported by NSF Grants (CCF-1703925, CCF-1421161, CCF-1714818, CCF-1617955 and CCF-1740833), Simons Foundation (#491119) and Google Research Award.

46th International Colloquium on Automata, Languages, and Programming (ICALP 2019).
Editors: Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi;
Article No. 14; pp. 14:1–14:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The success of modern parallel and distributed systems such as MapReduce [16, 17], Spark [41], Hadoop [39], Dryad [23], together with the need to solve problems on massive data, is driving the development of new algorithms which are more efficient and scalable in these large-scale systems. An important theoretical problem is to develop models which are good abstractions of these computational frameworks. The *Massively Parallel Computation (MPC)* model [25, 21, 11, 3, 9, 15, 4] captures the capabilities of these computational systems while keeping the description of the model itself simple. In the MPC model, there are machines (processors), each with $\Theta(N^\delta)$ local memory, where N denotes the size of the input and $\delta \in (0, 1)$. The computation proceeds in rounds, where each machine can perform unlimited local computation in a round and exchange $O(N^\delta)$ data at the end of the round. The parallel time of an algorithm is measured by the total number of computation-communication rounds. The MPC model is a variant of the Bulk Synchronous Parallel (BSP) model [38]. It is also a more powerful model than the PRAM since any PRAM algorithm can be simulated in the MPC model [25, 21] while some problem can be solved in a faster parallel time in the MPC model. For example, computing the XOR of N bits takes $O(1/\delta)$ parallel time in the MPC model but needs near-logarithmic parallel time on the most powerful CRCW PRAM [10].

A natural question to ask is: which problems can be solved in faster parallel time in the MPC model than on a PRAM? This question has been studied by a line of recent papers [25, 19, 29, 3, 1, 6, 22, 15, 7, 14, 13, 32, 20]. Most of these results studied the graph problems, which are the usual benchmarks of parallel/distributed models. Many graph problems such as graph connectivity [35, 33, 30], graph biconnectivity [37, 36], maximal matching [26], minimum spanning tree [27] and maximal independent set [31, 2] can be solved in the standard logarithmic time in the PRAM model, but these problems have been shown to have a better parallel time in the MPC model.

In addition, we hope to develop *fully scalable* algorithms for the graph problems, i.e., the algorithm should work for any constant $\delta > 0$. The previous literatures show that a graph problem in the MPC model with large local memory size may be much easier than the same problem in the MPC model but with a smaller local memory size. In particular, when the local memory size per machine is close to the number of vertices n, many graph problems have efficient algorithms. For example, if the local memory size per machine is $n/\log^{O(1)} n$, the connectivity problem [7] and the approximate matching problem [5] can be solved in $O(\log \log n)$ parallel time. If the local memory size per machine is $\Omega(n)$, then the MPC model meets the congested clique model [12]. In this setting, the connectivity problem and the minimum spanning tree problem can be solved in $O(1)$ parallel time [24]. If the local memory size per machine is $n^{1+\Omega(1)}$, many graph problems such as maximal matching, approximate weighted matchings, approximate vertex and edge covers, minimum cuts, and the biconnectivity problem can be solved in $O(1)$ parallel time [29, 8]. The landscape of graph algorithms in the MPC model with small local memory is more nuanced and challenging for algorithm designers. If the local memory size per machine is $n^{1-\Omega(1)}$, then the best connectivity algorithm takes parallel time $O(\log D \log \log n)$ where D is the diameter of the graph [4], and the best approximate maximum matching algorithm takes parallel time $\widetilde{O}(\sqrt{\log n})$ [32].

Therefore, the main open question is: which kind of the graph problems can have faster fully scalable MPC algorithms than the standard logarithmic PRAM algorithms?

Two fundamental graph problems in graph theory are 2-edge connectivity and 2-vertex connectivity (biconnectivity). In this work, we studied these two problems in the MPC model. 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 an equivalence relation [18]. 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. We proposed faster, fully scalable algorithms for the both 2-edge connectivity problem and the biconnectivity problem by parameterizing the running time as a function of the *diameter* and the *bi-diameter* of the graph. The diameter D of G is the largest diameter of its connected components. 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. Our main results are 1) a fully scalable $O(\log D \log \log_{m/n} n)$ parallel time 2-edge connectivity algorithm, 2) a fully scalable $O(\log D \log^2 \log_{m/n} n + \log D' \log \log_{m/n} n)$ parallel time biconnectivity algorithm. Our 2-edge connectivity algorithm achieves the same parallel time as the connectivity algorithm of [4]. We also show an $\Omega(\log D')$ conditional lower bound for the biconnectivity problem.

1.1 The Model

Our model of computation is the Massively Parallel Computation (MPC) model [25, 21, 11].

Consider two non-negative parameters $\gamma \geq 0, \delta > 0$. In the (γ, δ)-MPC model [4], there are p machines (processors) each with local memory size s, where $p \cdot s = \Theta(N^{1+\gamma}), s = \Theta(N^{\delta})$ and N denotes the size of the input data. Thus, the space per machine is sublinear in N, and the total space is only an $O(N^{\gamma})$ factor more than the input size. In particular, if $\gamma = 0$, the total space available in the system is linear in the input size N. The space size is measured by words each containing $\Theta(\log(s \cdot p))$ bits. Before the computation starts, the input data is distributed on $\Theta(N/s)$ input machines. The computation proceeds in rounds. In each round, each machine can perform local computation on its local data, and send messages to other machines at the end of the round. In a round, the total size of messages sent/received by a machine should be bounded by its local memory size $s = \Theta(N^{\delta})$. For example, a machine can send s size 1 messages to s machines or send a size s message to 1 machine in a single round. However, it cannot broadcast a size s message to every machine. In the next round, each machine only holds the received messages in its local memory. At the end of the computation, the output data is distributed on the output machines. An algorithm in this model is called a (γ, δ)-MPC algorithm. The parallel time of an algorithm is the total number of rounds needed to finish its computation. In this paper, we consider δ an arbitrary constant in $(0, 1)$.

1.2 Our Results

Our main results are efficient MPC algorithms for 2-edge connectivity and biconnectivity problems. In our algorithms, one important subroutine is computing the Depth-First-Search (DFS) sequence [4] which is a variant of the Euler tour representation proposed by [37, 36] in 1984. We show how to efficiently compute the DFS sequence in the MPC model with linear total space. Conditioned on the hardness of the connectivity problem in the MPC model, we prove a hardness result on the biconnectivity problem.

For 2-edge connectivity and biconnectivity, the input is an undirected graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. $N = n + m$ denotes the size of the representation of G, D denotes the diameter of G, and D' denotes the bi-diameter of G. We state our results in the following.

Biconnectivity. In the biconnectivity problem, we want to find all the 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 (x, c) with $x = e$ 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.

▶ **Theorem 1** (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.*

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.

2-Edge connectivity. In the 2-edge connectivity problem, we want to output all the bridges of the input graph G. Since an edge is a bridge if and only if each of its endpoints is either a cut vertex or a vertex with degree 1, the 2-edge connectivity problem should be easier than the biconnectivity problem. We show how to solve 2-edge connectivity in the same parallel time as the algorithm proposed by [4] for solving connectivity.

▶ **Theorem 2** (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.*

DFS sequence. A rooted tree with a vertex set V can be represented by $n = |V|$ pairs $(v_1, \text{par}(v_1)), (v_2, \text{par}(v_2)), \cdots, (v_n, \text{par}(v_n))$ where $\text{par} : V \to V$ is a set of parent pointers, i.e., for a non-root vertex v, $\text{par}(v)$ denotes the parent of v, and for the root vertex v, $\text{par}(v) = v$. We show an algorithm which can compute the DFS sequence (Definition 6) of the rooted tree in the MPC model with linear total space.

▶ **Theorem 3** (DFS sequence of a tree in MPC). *Given a rooted tree represented by a set of parent pointers $\text{par} : V \to V$, there is a randomized $(0, \delta)$-MPC algorithm which outputs the DFS sequence in $O(\log D)$ parallel time, where $\delta \in (0, 1)$ is an arbitrary constant, D is the depth of the tree. The success probability is at least 0.99. If the algorithm fails, then it returns FAIL.*

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 [25, 11, 28, 34, 40]. Under this conjecture, we show that $\Omega(\log D')$ parallel time is necessary for the biconnectivity problem, and this is true even when $D = O(1)$, i.e., the diameter of the graph is a constant.

▶ **Theorem 4** (Hardness of biconnectivity in MPC). *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.*

1.3 Our Techniques

Biconnectivity. At a high level our biconnectivity algorithm is based on a framework proposed by [36]. The main idea is to construct a new graph and reduce the problem of finding biconnected components of G to the problem of finding connected components of the new graph G'. At first glance, it should be efficiently solved by the connectivity algorithm [4]. However, there are two main issues: 1) since the parallel time of the MPC connectivity algorithm of [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 [36]. Without loss of generality, we can suppose the input $G = (V, E)$ is connected. 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 [18], 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 $O(k \cdot d)$. Thus, according to [4], we can find the connected components of G' in $\sim (\log k + \log d)$ parallel time, where d and k are upper bounded by the diameter and the bi-diameter of G respectively.

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 $\mathrm{lev}(x)$ be the minimum depth of the least 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, [36] 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. Instead, we follow the leaf sampling framework proposed by [4]. Although the DFS sequence algorithm proposed by [4] takes $O(\log d)$ time where d is the depth of T, it needs $\Omega(n \log d)$ total space. The bottleneck is the subroutine which needs to solve the least common ancestors problem and generate multiple path sequences. The previous algorithm uses the doubling algorithm for the subroutine, i,e., for each vertex v, they store the 2^i-th ancestor of v for every $i \in [\lceil \log d \rceil]$. This is the reason why [4] cannot achieve the linear total space. We 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.

2-Edge connectivity. Without loss of generality, we can assume the input graph G is connected. 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 on the outside of the subtree of u. Similarly, the second case can be solved by some range queries over the DFS sequence of T.

Conditional hardness for biconnectivity. We want to reduce the connectivity problem to the biconnectivity problem. For an undirected graph G, if we add an additional vertex v^* and connects v^* to every vertex of G, then the diameter of the resulting graph G' is at most 2 and each biconnected components of G' corresponds to a connected component of G. Furthermore, the bi-diameter of G' is upper bounded by the diameter of G plus 2. Therefore, if the parallel time of an algorithm $\mathcal{A}'$ for finding the biconnected components of G' depends on the bi-diameter of G', there exists an algorithm $\mathcal{A}$ which can find all the connected components of G in the parallel time which has the same dependence on the diameter of G.

1.4 A Roadmap

Section 2 introduces the notation and some useful definitions. Section 3 describes the offline algorithms for 2-edge connectivity and biconnectivity. It also includes some crucial properties of the algorithms. In Section 4, we show an linear space offline algorithm to find the DFS sequence of a tree. All of these offline algorithms can be implemented in the MPC model efficiently. Section 5 contains the conditional hardness result for the biconnectivity problem in the MPC model. For the MPC implementations and all the missing technical proofs, we refer readers to the full version of the paper.

2 Preliminaries

2.1 Notation

We follow the notation of [4]. $[n]$ denotes the set of integers $\{1, 2, \cdots, n\}$.

Diameter and bi-diameter. Consider an undirected graph G with a vertex set V and an edge set E. For any two vertices u, v, we use $\text{dist}_G(u, v)$ to denote the distance between u and v in graph G. If u, v are not in the same (connected) component of G, then $\text{dist}_G(u, v) = \infty$. The diameter $\text{diam}(G)$ of G is the largest diameter of its connected components, i.e., $\text{diam}(G) = \max_{u,v \in V: \text{dist}_G(u,v) \neq \infty} \text{dist}_G(u, v)$. $(v_1, v_2, \cdots, v_k) \in V^k$ 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 $\text{cyclen}_G(u, v)$ to denote the minimum length of a simple cycle which contains both vertices u and v. If there is no simple cycle which contains both u and v, $\text{cyclen}_G(u, v) = \infty$. $\text{cyclen}_G(u, u)$ is defined as 0. The bi-diameter of G, $\text{bi-diam}(G)$, is defined as $\max_{u,v \in V: \text{cyclen}_G(u,v) \neq \infty} \text{cyclen}_G(u, v)$.

Representation of a rooted forest. Let V denote a set of vertices. We represent a rooted forest in the same manner as [4]. Consider a mapping $\text{par} : V \to V$. For $i \in \mathbb{N}_{>0}$ and $v \in V$, we define $\text{par}^{(i)}(v)$ as $\text{par}(\text{par}^{(i-1)}(v))$, and $\text{par}^{(0)}(v)$ is defined as v itself. If $\forall v \in V, \exists i > 0$ such that $\text{par}^{(i)}(v) = \text{par}^{(i+1)}(v)$, then we call 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. Notice that par actually can represent a rooted forest, thus par can have more than one root. The depth of $v \in V$, $\text{dep}_{\text{par}(v)}$ is the smallest $i \in \mathbb{N}$ such that $\text{par}^{(i)}(v)$ is the same as $\text{par}^{(i+1)}(v)$. 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, $\text{dep}(\text{par})$ is defined as $\max_{v \in V} \text{dep}_{\text{par}}(v)$.

Ancestor and path. For two vertices $u, v \in V$, if $\exists i \in \mathbb{N}$ such that $u = \text{par}^{(i)}(v)$, then u is an ancestor of v (in par). If u is an ancestor of v, then the path $P(v, u)$ (in par) from v to u is a sequence $(v, \text{par}(v), \text{par}^{(2)}(v), \cdots, u)$ and the path $P(u, v)$ is the reverse of $P(v, u)$, i.e., $P(u, v) = (u, \cdots, \text{par}^{(2)}(v), \text{par}(v), v)$. If an ancestor u of v is also an ancestor of w, then u is a common ancestor of (v, w). Furthermore, if a common ancestor u of (v, w) satisfies $\text{dep}_{\text{par}}(u) \geq \text{dep}_{\text{par}}(x)$ for any common ancestor x of (v, w), then u is the lowest common ancestor (LCA) of (v, w).

Children and leaves. For any non-root vertex u of par, u is a child of $\text{par}(u)$. For any vertex $v \in V$, $\text{child}_{\text{par}}(v)$ denotes the set of all the children of v, i.e., $\text{child}_{\text{par}}(v) = \{u \in V \mid u \neq v, \text{par}(u) = v\}$. If u is the k^{th} smallest vertex in the set $\text{child}_{\text{par}}(v)$, then we define $\text{rank}_{\text{par}}(u) = k$, or in other words, u is the k^{th} child of v. If v is a root vertex of par, then $\text{rank}_{\text{par}}(v)$ is defined as 1. $\text{child}_{\text{par}}(v, k)$ denotes the k^{th} child of v. For simplicity, if par is clear in the context, we just use $\text{child}(v)$, $\text{rank}(v)$ and $\text{child}(v, k)$ to denote $\text{child}_{\text{par}}(v)$, $\text{rank}_{\text{par}}(v)$ and $\text{child}_{\text{par}}(v, k)$ for short. If $\text{child}(v) = \emptyset$, then v is a leaf of par. We denote $\text{leaves}(\text{par})$ as the set of all the leaves of par, i.e., $\text{leaves}(\text{par}) = \{v \mid \text{child}(v) = \emptyset\}$.

2.2 Depth-First-Search Sequence

The Euler tour representation of a tree is proposed by [37, 36]. It is a crucial building block in many graph algorithms including biconnectivity algorithms. The Depth-First-Search (DFS) sequence [4] of a rooted tree is a variant of the Euler tour representation. Let us first introduce some relevant concepts of the DFS sequence.

▶ **Definition 5** (Subtree [4]). *Consider a set of parent pointers* $\mathrm{par} : V \to V$ *on a vertex set* V*. Let* v *be a vertex in* V*, and let* $V' = \{u \in V \mid v \text{ is an ancestor of } u\}$*.* $\mathrm{par}' : V' \to V'$ *is 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* par′ *is a subtree of* v *in* par*. For* $u \in V'$*, we say* u *is in the subtree of* v*.*

The definition of the DFS sequence is the following:

▶ **Definition 6** (DFS sequence [4]). *Consider a set of parent pointers* $\mathrm{par} : V \to V$ *on a vertex set* V*. Let* v *be a vertex in* V*. If* v *is a leaf in* par*, then the DFS sequence of the subtree of* v *is* (v)*. Otherwise, the DFS sequence of the subtree of* v *is defined recursively 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 $k = |\mathrm{child}(v)|$ *and* $\forall i \in [k]$*,* $(a_{i,1}, a_{i,2}, \cdots, a_{i,n_i})$ *is the DFS sequence of the subtree of* $\mathrm{child}(v, i)$*, i.e., the* i^{th} *child of* v*.*

If $\mathrm{par} : V \to V$ has a unique root v, then we define the DFS sequence of par as the DFS sequence of the subtree of v. By the definition of the DFS sequence, for any two consecutive elements a_i and a_{i+1} in the sequence, a_i is either a parent of a_{i+1} or a_i is a child of a_{i+1}. Furthermore, for any vertex v, if both elements a_i and a_j $(i < j)$ in the DFS sequence A are v, any element a_k between a_i and a_j (i.e., $i \leq k \leq j$) should be a vertex in the subtree of v.

3 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 offline setting.

3.1 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.

▶ **Lemma 7** (2-Edge connectivity). *Consider an undirected graph* $G = (V, E)$*. Let* B *be the output of* Bridges(G)*. Then* B *is the set of all the bridges of* G*.*

3.2 Biconnectivity

In this section, we will show a biconnectivity algorithm. It is a modification of the algorithm proposed by [36]. 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 the connectivity algorithm [4] depends on the diameter of the graph, we also give an analysis of the diameter of the graph G'.

Algorithm 1 2-Edge Connectivity Algorithm.

- **Input**:
  - A connected undirected graph $G = (V, E)$.
- **Output**:
  - A subset of edges $B \subseteq E$.
- **Finding bridges** (BRIDGES($G = (V, E)$)):
  1. Compute a rooted spanning tree of G. The spanning tree is represented by a set of parent pointers $\text{par} : V \to V$.
  2. Compute $\text{lev} : V \to \mathbb{Z}_{\geq 0}$: for each $v \in V$,
  $$\text{lev}(v) \leftarrow \min\left(\text{dep}_{\text{par}}(v), \min_{w \in V \setminus \{\text{par}(v)\} : (v,w) \in E} \text{dep}_{\text{par}}(\text{the LCA of } (v, w))\right).$$
  3. Compute the DFS sequence A of par.
  4. 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.

Algorithm 2 Biconnectivity Algorithm.

- **Input**:
  - A connected undirected graph $G = (V, E)$.
- **Output**:
  - A coloring $\text{col} : E \to V$ of the edges.
- **Finding blocks** (BICONN($G = (V, E)$)):
  1. Compute a rooted spanning tree of G. The spanning tree is represented by a set of parent pointers $\text{par} : V \to V$.
  2. Compute $\text{lev} : V \to \mathbb{Z}_{\geq 0}$: for each $v \in V$,
  $$\text{lev}(v) \leftarrow \min\left(\text{dep}_{\text{par}}(v), \min_{w \in V \setminus \{\text{par}(v)\} : (v,w) \in E} \text{dep}_{\text{par}}(\text{the LCA of } (v, w))\right).$$
  3. Compute the DFS sequence A of par.
  4. Let r be the root of par. Initialize $V' \leftarrow V \setminus \{r\}, E' \leftarrow \emptyset$.
  5. For each $v \in V'$, let a_i, a_j be the first and the last appearance of v in A respectively. If $\min_{k \in \{i, i+1, \cdots, j\}} \text{lev}(a_k) < \text{dep}_{\text{par}}(\text{par}(v))$, $E' \leftarrow E' \cup \{(v, \text{par}(v))\}$.
  6. For each $(u, v) \in E$, if neither u nor v is the LCA of (u, v) in par, $E' \leftarrow E' \cup \{(u, v)\}$.
  7. Compute the connected components of $G' = (V', E')$. Let $\text{col}' : V' \to V'$ be the coloring of the vertices in V' such that $\forall u', v' \in V'$, u', v' are in the same connected component in $G' \Leftrightarrow \text{col}'(u') = \text{col}'(v')$.
  8. Initialize $\text{col} : E \to V$. For each $e = (u, v) \in E$, if $\text{dep}_{\text{par}}(u) \geq \text{dep}_{\text{par}}(v)$, set $\text{col}(e) \leftarrow \text{col}'(u)$; otherwise, set $\text{col}(e) \leftarrow \text{col}'(v)$. Output $\text{col} : E \to V$.

▶ **Lemma 8** (Biconnectivity). *Consider an undirected graph $G = (V, E)$. Let* $\mathrm{col} : E \to V$ *be the output of* $\textsc{Biconn}(G)$. *Then* $\forall e, e' \in E, e \neq e'$, col *satisfies* $\mathrm{col}(e) = \mathrm{col}(e') \Leftrightarrow$ *there is a simple cycle in G which contains both e and e'. Furthermore, the diameter of the graph G' constructed by* $\textsc{Biconn}(G)$ *is at most* $O(\mathrm{dep}(\mathrm{par}) \cdot \text{bi-diam}(G))$, *the number of vertices of G' is at most $|V|$, and the number of edges of G' is at most $|E|$.*

Algorithm 3 Leaf Sampling Algorithm for DFS Sequence.

- **Pre-determined**:
  - A threshold value s. //s will be the local memory size in the MPC model.
- **Input**:
  - A rooted tree represented by a set of parent pointers $\mathrm{par} : V \to V$ on a set V of n vertices (i.e., par has a unique root r).
- **Output**:
  - The DFS sequence of the rooted tree represented by par.
- **Leaf sampling algorithm** ($\textsc{LeafSampling}(s, \mathrm{par} : V \to V)$):

1. If $n \leq s$, return the DFS sequence of par directly.
2. Set $t \leftarrow \Theta(s^{1/3} \log n)$, $L \leftarrow \mathrm{leaves}(\mathrm{par})$.
3. Each $v \in L$ is independently chosen with probability $p = \min(1, t/|L|)$, and let $S = \{l_1, l_2, \cdots, l_k\}$ be the set of samples. If $|S|^2 > s$, output FAIL.
4. For every pair of sampled leaves $x, y \in S$ with $x \neq y$, find the least common ancestor $p_{x,y}$ of (x, y), and set $p_{xy,x}$, $p_{xy,y}$ to be two children of $p_{x,y}$ such that $p_{xy,x}$ is an ancestor of x and $p_{xy,y}$ is an ancestor of y.
5. Sort $l_1, l_2, \cdots, l_k \in S$ such that $\forall i < j \in [k]$, $\mathrm{rank}(p_{l_i l_j, l_i}) < \mathrm{rank}(p_{l_i l_j, l_j})$.
6. Find the paths $A'_1 = P(r, l_1), A'_2 = P(\mathrm{par}(l_1), p_{l_1,l_2}), A'_3 = P(p_{l_1 l_2, l_2}, l_2), \cdots, A'_{2k-2} = P(\mathrm{par}(l_{k-1}), p_{l_{k-1} l_k}), A'_{2k-1} = P(p_{l_{k-1} l_k, l_k}, l_k), A'_{2k} = P(l_{2k}, r)$, i.e., the paths: $r \to l_1 \to$ the LCA of $(l_1, l_2) \to l_2 \to \cdots \to l_{k-1} \to$ the LCA of $(l_{k-1}, l_k) \to l_k \to r$.
7. Set $A' \leftarrow A'_1 A'_2 \cdots A'_{2k}$, i.e., A' is the concatenation of $A'_1, A'_2, \cdots, A'_{2k}$.
8. For each element a'_i in the i^{th} $(i > 1)$ position of the sequence A',
   - if the vertex a'_i is a leaf, keep a'_i as a single copy;
   - Otherwise,
     - if $a'_{i-1} = \mathrm{par}(a'_i)$, i.e., i is the first position that the vertex a'_i appears in A', split a'_i into $\mathrm{rank}(a'_{i+1})$ copies; //a'_{i+1} is a child of a'_i.
     - if $a'_{i-1}, a'_{i+1} \in \mathrm{child}(a'_i)$, split a'_i into $\mathrm{rank}(a'_{i+1}) - \mathrm{rank}(a'_{i-1})$ copies;
     - if $a'_{i+1} = \mathrm{par}(a'_i)$, i.e., i is the last position that the vertex a'_i appears in A', split a'_i into $|\mathrm{child}(a'_i)| - \mathrm{rank}(a'_{i-1})$ copies. //a'_{i-1} is a child of a'_i.

   Let A'' be the result sequence.
9. For each $v \in V$, if $\mathrm{par}(v)$ appears in A'' but v does not appear in A'', recursively find the DFS sequence of the subtree of v, and insert the such sequence into the position after the $\mathrm{rank}(v)^{\text{th}}$ appearance of $\mathrm{par}(v)$ in A''. Output the final result sequence A.

4 An Offline DFS Sequence Algorithm in Linear Space

In Section 4.1, we will review an algorithmic framework proposed by [4] for the DFS sequence. In Section 4.2, 4.3, 4.4, we will discuss the subroutines needed for our DFS sequence algorithm in the offline setting.

4.1 DFS Sequence via Leaf Sampling

In the following, we review the leaf sampling algorithmic framework proposed by [4] for finding the DFS sequence of a rooted tree.

▶ **Theorem 9** (Leaf sampling algorithm [4]). *Consider a set of parent pointers* $\text{par} : V \to V$ *on a set* V *of* n *vertices. Suppose* par *has a unique root. For any* $\gamma \geq 0$ *and any constant* $\delta \in (0, 1)$*, if both of step 4 and step 6 in* LeafSampling(n^δ, par) *can be implemented in the* (γ, δ)*-MPC model with* $O(\log(\text{dep}(\text{par})))$ *parallel time, then the leaf sampling algorithm with parameter* $s = n^\delta$ *on input* $\text{par} : V \to V$ *can be implemented in the* (γ, δ)*-MPC model. Furthermore, with probability at least* 0.99, LeafSampling(n^δ, par) *can output the DFS sequence of* par *in* $O(\log(\text{dep}(\text{par})))$ *parallel time. If the algorithm fails, then it returns FAIL.*

By Theorem 9, we only need to give a linear total space MPC algorithm for the LCA problem and the path generation problem to design an efficient DFS sequence algorithm in the $(0, \delta)$-MPC model.

In [4], they proposed to use doubling algorithms to compute the LCA and generate the paths. Since they need to store the every 2^i-th ancestor for each vertex, the total space needed is $\Theta(n \cdot \log(\text{the depth of the tree}))$. We show that we only need to apply the doubling algorithm for a compressed tree, instead of applying it for the original tree.

Algorithm 4 Construction of a Compressed Rooted Tree.

- **Input**:
  - A rooted tree represented by a set of parent pointers $\text{par} : V \to V$ on a set V of n vertices (par has a unique root r).
- **Output**:
  - A vertex set $V' \subseteq V$, a set of parent pointers $\text{par}' : V' \to V'$ on V'.
- **Tree compression** (Compress$(\text{par} : V \to V)$):
  1. Compute the depth of par, the depth of each vertex and set $d \leftarrow \text{dep}(\text{par})$, $t \leftarrow \lceil \log d \rceil$.
  2. $V' \leftarrow \{v \in V \mid \text{dep}_{\text{par}}(v) \mod t = 0, \text{dep}_{\text{par}}(v) + t \leq d\}$.
  3. Initialize $\text{par}' : V' \to V'$. For each $v \in V'$, $\text{par}'(v) \leftarrow \text{par}^{(t)}(v)$.
  4. Output V', par′.

4.2 Compressed Rooted Tree

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

▶ **Lemma 10** (Properties of a compressed rooted tree). *Let* $\text{par} : V \to V$ *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* $(V', \text{par}') =$Compress(par)*. Then it has the following properties:*

1. $|V'| \leq |V| / \log(\text{dep}(\text{par}))$.
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* $\text{par}^{(i)}(v) \in V'$.

4.3 Least Common Ancestor

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 $\forall i \in [q], u_i \neq v_i, u_i, v_i \in \text{leaves}(\text{par})$, we show a space efficient algorithm which can output the LCA of each queried

Algorithm 5 Lowest Common Ancestor.

- **Input**:
  - A rooted tree represented by a set of parent pointers $\text{par} : V \to V$ on a set V of n vertices (par has a unique root r), and a set of q queries $Q = \{(u_1, v_1), (u_2, v_2), \cdots, (u_q, v_q)\}$ where $\forall i \in [q], u_i \neq v_i, u_i, v_i \in \text{leaves(par)}$.
- **Output**:
  - $\text{lca} : Q \to V \times V \times V$.
- **Finding LCA** ($\text{LCA}(\text{par} : V \to V, Q)$):
  1. $(V', \text{par}') \leftarrow$COMPRESS(par). //(see Lemma 10).
  2. Set $d \leftarrow \text{dep(par)}, t \leftarrow \lceil \log d \rceil$ and compute mappings $g_0, g_1, \cdots g_t : V' \to V'$ such that $\forall v \in V', j \in \{0, 1, \cdots, t\}$, $g_j(v) = \text{par}'^{(2^j)}(v)$.
  3. For each query $(u_i, v_i) \in Q$: //Suppose $\text{dep}_{\text{par}}(u_i) \geq \text{dep}_{\text{par}}(v_i)$.
     - **a.** If $\text{dep}_{\text{par}}(u_i) > \text{dep}_{\text{par}}(v_i)+2t$, find an ancestor $\widehat{u}_i$ of u_i in par such that $\text{dep}_{\text{par}}(\widehat{u}_i) \leq \text{dep}_{\text{par}}(v_i) + 2t$ and $\text{dep}_{\text{par}}(\widehat{u}_i) \geq \text{dep}_{\text{par}}(v_i)$. Otherwise, $\widehat{u}_i \leftarrow u_i$.
     - **b.** If $\exists j \in [4t]$ $\text{par}^{(j)}(\widehat{u}_i)$ is the LCA of $(\widehat{u}_i, v_i)$ in par, set $\text{lca}(u_i, v_i) = (\text{par}^{(j)}(\widehat{u}_i), x, y)$ where x, y are children of $\text{par}^{(j)}(\widehat{u}_i)$ and x, y are ancestors of $\widehat{u}_i, v_i$ respectively. The query of (u_i, v_i) is finished.
     - **c.** 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., $\text{dep}_{\text{par}}(\widehat{u}_i) - \text{dep}_{\text{par}}(u'_i)$ 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., $\text{dep}_{\text{par}}(v_i) - \text{dep}_{\text{par}}(v'_i)$ is minimized.
     - **d.** Find $u''_i \neq v''_i \in V'$ such that they are ancestors of u'_i and v'_i respectively, and $\text{par}'(u''_i) = \text{par}'(v''_i)$ is the LCA of (u'_i, v'_i) in par$'$.
     - **e.** Find the smallest $j \in [2t]$ such that $\text{par}^{(j)}(u''_i) = \text{par}^{(j)}(v''_i)$. Set $\text{lca}(u_i, v_i) = (\text{par}^{(j)}(u''_i), \text{par}^{(j-1)}(u''_i), \text{par}^{(j-1)}(v''_i))$.

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.

Before we analyze the algorithm $\text{LCA}(\text{par}, Q)$, let us discuss some details of the algorithm.

1. We pre-compute $\text{dep}_{\text{par}}(v)$ and $\text{dep}_{\text{par}'}(u)$ for every $v \in V$ and $u \in V'$.
2. To implement step 3a, we firstly check whether $\text{dep}_{\text{par}}(u_i) > \text{dep}_{\text{par}}(v_i) + 2t$. If it is not true, we can set $\widehat{u}_i$ to be u_i directly. Otherwise, according to Lemma 10, there is a $j \in \{0, 1, \cdots, 2t\}$ such that $\text{par}^{(j)}(u_i) \in V'$. Since $\text{dep}_{\text{par}}(u_i) > \text{dep}_{\text{par}}(v_i) + 2t$, $\text{dep}_{\text{par}}(\text{par}^{(j)}(u_i)) > \text{dep}_{\text{par}}(v_i)$. We initialize $\widehat{u}_i$ to be $\text{par}^{(j)}(u_i) \in V'$. For $k = t \to 0$, if $\text{dep}_{\text{par}}(g_k(\widehat{u}_i)) > \text{dep}_{\text{par}}(v_i)$ (i.e., $\text{dep}_{\text{par}}(\text{par}'^{(2^k)}(\widehat{u}_i)) > \text{dep}_{\text{par}}(v_i)$), we set $\widehat{u}_i \leftarrow g_k(\widehat{u}_i) = \text{par}'^{(2^k)}(\widehat{u}_i)$. Due to Lemma 10 again, the final $\widehat{u}_i$ must satisfy $\text{dep}_{\text{par}}(\widehat{u}_i) \geq \text{dep}_{\text{par}}(v_i)$ and $\text{dep}_{\text{par}}(\widehat{u}_i) \leq \text{dep}_{\text{par}}(v_i) + 2t$. This step takes time $O(t)$.

▶ **Lemma 11** (LCA algorithm). *Let* $\text{par} : V \to V$ *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* $\forall i \in [q], u_i \neq v_i, u_i, v_i \in \text{leaves(par)}$. *Let* $\text{lca} : Q \to V \times V \times V$ *be the output of* $\text{LCA}(\text{par}, Q)$. *For* $(u_i, v_i) \in Q$, $(p_i, p_{i,u_i}, p_{i,v_i}) = \text{lca}(u_i, v_i)$ *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. *Furthermore, the space used by the algorithm is at most* $O(|Q| + |V|)$.

4.4 Multi-Paths Generation

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 show a space efficient algorithm MultiPaths(par, Q) which can generate all the paths $P(u_1, v_1), P(u_2, v_2), \cdots, P(u_q, v_q)$.

Algorithm 6 Multi-Paths Generation.

- **Input**:
  - A rooted tree represented by a set of parent pointers $\text{par} : V \to V$ on a set V of n vertices (par has a unique root r), 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.
- **Output**:
  - $P_1, P_2, \cdots, P_q$.
- **Generating multiple path sequences** (MultiPaths$(\text{par} : V \to V, Q)$):
  1. $(V', \text{par}') \leftarrow$Compress(par). //(see Lemma 10).
  2. Set $d \leftarrow \text{dep}(\text{par}), t \leftarrow \lceil \log d \rceil$ and compute mappings $g_0, g_1, \cdots g_t : V' \to V'$ such that $\forall v \in V', j \in \{0, 1, \cdots, t\}$, $g_j(v) = \text{par}'^{(2^j)}(v)$.
  3. For each vertex-ancestor pair $(u_i, v_i) \in Q$:
     a. 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.
     b. 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 $\text{dep}_{\text{par}}(v'_i) - 2t \leq \text{dep}_{\text{par}}(v_i)$.
     c. Generate the path $P'(u'_i, v'_i)$ in par$'$.
     d. Initialize a sequence A as the concatenation of (u_i), $P'(u'_i, v'_i)$ and (v_i).
     e. 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.

Before we analyze the correctness of the algorithm, let us discuss some details.

1. In step 3a, 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 $\text{par}^{(j)}(u_i) = \text{par}(\text{par}^{(j-1)}(u_i))$.
2. In step 3b, we want to find v'_i. We initialize v'_i as u'_i. For $k = t \to 0$, if $\text{dep}_{\text{par}}(g_k(v'_i)) > \text{dep}_{\text{par}}(v_i)$ (i.e., $\text{dep}_{\text{par}}(\text{par}'^{(2^k)}(v'_i)) > \text{dep}_{\text{par}}(v_i)$), we set $v'_i \leftarrow g_k(v'_i) = \text{par}'^{(2^k)}(v'_i)$.

▶ **Lemma 12** (Generation of multiple paths). *Let* $\text{par} : V \to V$ *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* MultiPaths(par, Q). *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. *Furthermore, the space used by the algorithm is at most* $O(|V| + \sum_{j \in [q]} |P_j|)$.

5 Hardness of Biconnectivity in MPC

There is a conjectured hardness which is widely used in the MPC literature [25, 11, 28, 34, 40].

▷ Conjecture 1 (1-cycle vs. 2-cycles). For any $\gamma \geq 0$ and any constant $\delta \in (0, 1)$, distinguishing the following two 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 13** (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 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' has 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 $\text{bi-diam}(G) = \Theta(n)$. If G' contains two cycles, then G contains two biconnected components and $\text{bi-diam}(G) = \Theta(n)$.

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 1) is false. ◀

References

1 Kook Jin Ahn and Sudipto Guha. Access to data and number of iterations: Dual primal algorithms for maximum matching under resource constraints. *ACM Transactions on Parallel Computing (TOPC)*, 4(4):17, 2018.

2 Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of algorithms*, 7(4):567–583, 1986.

3 Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 574–583. ACM, 2014.

4 Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. Parallel Graph Connectivity in Log Diameter Rounds, 2018. In *FOCS 2018*. `arXiv:1805.03055`.

5 Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab Mirrokni, and Cliff Stein. Coresets meet EDCS: algorithms for matching and vertex cover on massive graphs. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1616–1635. SIAM, 2019.

6 Sepehr Assadi and Sanjeev Khanna. Randomized composable coresets for matching and vertex cover. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 3–12. ACM, 2017.

7 Sepehr Assadi, Xiaorui Sun, and Omri Weinstein. Massively Parallel Algorithms for Finding Well-Connected Components in Sparse Graphs. *arXiv preprint*, 2018. `arXiv:1805.02974`.

8 Giorgio Ausiello, Donatella Firmani, Luigi Laura, and Emanuele Paracone. Large-scale graph biconnectivity in MapReduce. *Department of Computer and System Sciences Antonio Ruberti Technical Reports*, 4(4), 2012.

9 Boaz Barak, Jonathan A Kelner, and David Steurer. Dictionary learning and tensor decomposition via the sum-of-squares method. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing (STOC)*, pages 143–151. ACM, 2015. `arXiv:1407.1543`.

10 Paul Beame and Johan Hastad. Optimal bounds for decision problems on the CRCW PRAM. *Journal of the ACM (JACM)*, 36(3):643–670, 1989.

11 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 273–284. ACM, 2013.

12 Soheil Behnezhad, Mahsa Derakhshan, and MohammadTaghi Hajiaghayi. Brief announcement: Semi-mapreduce meets congested clique. *arXiv preprint*, 2018. `arXiv:1802.10297`.

13 Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, and Richard M Karp. Massively parallel symmetry breaking on sparse graphs: MIS and maximal matching. *arXiv preprint*, 2018. `arXiv:1807.06701`.

14 Sebastian Brandt, Manuela Fischer, and Jara Uitto. Matching and MIS for Uniformly Sparse Graphs in the Low-Memory MPC Model. *arXiv preprint*, 2018. `arXiv:1807.05374`.

15 Artur Czumaj, Jakub Łącki, Aleksander Mądry, Slobodan Mitrović, Krzysztof Onak, and Piotr Sankowski. Round compression for parallel matching algorithms. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 471–484. ACM, 2018.

16 Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *To appear in OSDI*, page 1, 2004.

17 Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

18 Reinhard Diestel. *Graph theory*. Springer Publishing Company, Incorporated, 2018.

19 Alina Ene, Sungjin Im, and Benjamin Moseley. Fast clustering using MapReduce. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 681–689. ACM, 2011.

20 Manuela Fischer, Mohsen Ghaffari, and Jara Uitto. Simple Graph Coloring Algorithms for Congested Clique and Massively Parallel Computation. *arXiv preprint*, 2018. `arXiv:1808.08419`.

21 Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, Searching, and Simulation in the MapReduce Framework. In *ISAAC*, volume 7074, pages 374–383. Springer, 2011.

22 Sungjin Im, Benjamin Moseley, and Xiaorui Sun. Efficient massively parallel methods for dynamic programming. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 798–811. ACM, 2017.

23 Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41 (3), pages 59–72. ACM, 2007.

24 Tomasz Jurdziński and Krzysztof Nowicki. MST in O(1) rounds of congested clique. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2620–2632. SIAM, 2018.

25 Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 938–948. Society for Industrial and Applied Mathematics, 2010.

26 Richard M Karp, Eli Upfal, and Avi Wigderson. Constructing a perfect matching is in random NC. *Combinatorica*, 6(1):35–48, 1986.

27 Valerie King, Chung Keung Poon, Vijaya Ramachandran, and Santanu Sinha. An optimal EREW PRAM algorithm for minimum spanning tree verification. *Information Processing Letters*, 62(3):153–159, 1997.

28 Raimondas Kiveris, Silvio Lattanzi, Vahab Mirrokni, Vibhor Rastogi, and Sergei Vassilvitskii. Connected components in mapreduce and beyond. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

29 Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 85–94. ACM, 2011.

30 Sixue Liu and Robert E Tarjan. Simple Concurrent Labeling Algorithms for Connected Components. *arXiv preprint*, 2018. `arXiv:1812.06177`.

31 Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

32 Krzysztof Onak. Round compression for parallel graph algorithms in strongly sublinear space. *arXiv preprint*, 2018. `arXiv:1807.08745`.

33 John H Reif. Optimal Parallel Algorithms for Interger Sorting and Graph Connectivity. Technical report, HARVARD UNIV CAMBRIDGE MA AIKEN COMPUTATION LAB, 1985.

34 Tim Roughgarden, Sergei Vassilvitskii, and Joshua 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*, pages 1–12. ACM, 2016.

35 Yossi Shiloach and Uzi Vishkin. An O (log n) parallel connectivity algorithm. Technical report, Computer Science Department, Technion, 1980.

36 Robert E Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.

37 Robert Endre Tarjan and Uzi Vishkin. Finding biconnected componemts and computing tree functions in logarithmic parallel time. In *25th Annual Symposium onFoundations of Computer Science, 1984.*, pages 12–20. IEEE, 1984.

38 Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

39 Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing (STOC)*, pages 887–898. ACM, 2012.

40 Grigory Yaroslavtsev and Adithya Vadapalli. Massively Parallel Algorithms and Hardness for Single-Linkage Clustering under Lp Distances. In *International Conference on Machine Learning*, pages 5596–5605, 2018.

41 Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.