Balliu, Alkida; Latypov, Rustam; Maus, Yannic; Olivetti, Dennis; Uitto, Jara

# Optimal Deterministic Massively Parallel Connectivity on Forests

*Published in:*
Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)

*DOI:*
10.1137/1.9781611977554.ch99

Published: 01/01/2023

*Document Version*
Publisher's PDF, also known as Version of record

# Optimal Deterministic Massively Parallel Connectivity on Forests*

Alkida Balliu
Gran Sasso Science Institute

Rustam Latypov†
Aalto University

Yannic Maus
TU Graz

Dennis Olivetti
Gran Sasso Science Institute

Jara Uitto
Aalto University



**Abstract**

We show fast deterministic algorithms for fundamental problems on forests in the challenging low-space regime of the well-known Massive Parallel Computation (MPC) model. A recent breakthrough result by Coy and Czumaj [STOC'22] shows that, in this setting, it is possible to deterministically identify connected components on graphs in $O(\log D + \log\log n)$ rounds, where $D$ is the diameter of the graph and $n$ the number of nodes. The authors left open a major question: is it possible to get rid of the additive $\log\log n$ factor and deterministically identify connected components in a runtime that is completely independent of $n$?

We answer the above question in the affirmative in the case of forests. We give an algorithm that identifies connected components in $O(\log D)$ deterministic rounds. The total memory required is $O(n+m)$ words, where $m$ is the number of edges in the input graph, which is optimal as it is only enough to store the input graph. We complement our upper bound results by showing that $\Omega(\log D)$ time is necessary even for component-unstable algorithms, conditioned on the widely believed 1 vs. 2 cycles conjecture. Our techniques also yield a deterministic forest-rooting algorithm with the same runtime and memory bounds.

Furthermore, we consider Locally Checkable Labeling problems (LCLs), whose solution can be verified by checking the $O(1)$-radius neighborhood of each node. We show that any LCL problem on forests can be solved in $O(\log D)$ rounds with a canonical deterministic algorithm, improving over the $O(\log n)$ runtime of Brandt, Latypov and Uitto [DISC'21]. We also show that there is no algorithm that solves all LCL problems on trees asymptotically faster.

## 1 Introduction

Graphs offer a versatile abstraction to relational data and there is a growing demand for processing graphs at scale. One of the most central graph problems in massive graph processing is the detection of connected components of the input graph. This problem both captures challenges in the study of the fundamentals of parallel computing and has a variety of practical applications. In this work, we introduce new parallel techniques for finding connected components of a graph. Furthermore, we show that our techniques can be applied to solve a broad family of other central graph problems.

The Massively Parallel Computation (MPC) model [29] is a mathematical abstraction of modern frameworks of parallel computing such as Hadoop [38], Spark [39], MapReduce [25], and Dryad [28]. In the MPC model, we have $M$ machines that communicate in synchronous rounds. In each round, every machine receives the messages sent in the previous round, performs (arbitrary) local computations, and is allowed to send messages to any other machine. Initially, an input graph of $n$ nodes and $m$ edges is distributed among the machines. At the end of the computation, each machine needs to know the output of each node it holds, e.g., the identifier of its connected component. We work in the low-space regime, where the *local memory* $S$ of each machines is limited to $n^\delta$ words of $O(\log n)$ bits, where $0 < \delta < 1$. A word is enough to store a node or a machine identifier from a polynomial (in $n$) domain. The local memory restricts the amount of data a machine initially holds and is allowed to send and receive per round. Furthermore, we focus on the most restricted case of *linear total memory*, i.e., $S \cdot M = \Theta(n+m)$. Notice that $\Omega(n+m)$ words are required to store the input graph.

In recent years, identifying connected components of a graph has gained a lot of attention. As a baseline, the widely believed 1 vs. 2 cycles conjecture states that it takes $\Omega(\log n)$ rounds to tell whether the input graph is a

*The full version of the paper can be accessed at `https://arxiv.org/abs/2211.03530`

†Supported by the Academy of Finland, Grant 334238

cycle of $n$ nodes or two cycles with $n/2$ nodes [36, 26, 14]. We note that proving any unconditional lower bounds seems out of reach as any non-constant lower bound in the low-space MPC model for any problem in P would imply a separation between NC[1] and P [36]. It has been shown that this conjecture also implies conditional hardness of detecting connected components in time $o(\log D)$ on the family graphs with diameter at most $D$ [14, 22].

This bound has been almost matched in a sequence of works. First, a randomized $O(\log D \cdot \log\log_{m/n} n)$ time algorithm was designed in [2]. This was further improved to $O(\log D + \log\log_{m/n} n)$ in [14] and derandomized with the same asymptotic runtime in [22]. All of the aforementioned algorithms require only $O(n+m)$ words of global memory. A fundamental question is whether the runtime *necessarily* depends on $n$ for some range of $m$; we give evidence towards a negative answer. We show that in the case of forests, we can identify the connected components of a graph in $O(\log D)$ time, which we show to be *optimal* under the 1 vs. 2 cycles conjecture.

**Connected Components on Forests.** Consider the family of forests with component-wise maximum diameter $D$. There is a deterministic low-space MPC algorithm to find the connected components in time $O(\log D)$. The algorithm uses $O(n+m)$ global memory. Under the 1 vs. 2 cycles conjecture, this is optimal.

**Sparsification and Dependence on $n$.** In previous works on connected components, the algorithms have an inherent dependency on the total number of nodes $n$ in the input graph. There is a technical reason for this dependency, also in the context of problems beyond connected components. A common algorithm design pattern is to first *sparsify* the input graph, i.e., the graph is made much smaller and the problem is solved in the sparser instance [2, 27, 24, 23]. Then, it is shown that a solution to the original input can be recovered from a solution on the sparsified graph. As an example, a method to sparsify graphs for connectivity is to perform node/edge contractions, that make the graph smaller and preserve connectivity.

In this pattern, the denser the input graph is, the more global memory the algorithm has on the sparsified graph, relatively speaking. In the aforementioned previous works, the base $m/n$ of the logarithm can be replaced by $F/n$, where $F$ is the global memory. Hence, if $F = n^{1+\Omega(1)}$, the dependency on $n$ disappears. This suggests that the hardest instances are *sparse* graphs, as the $n$ dependency in the runtime of $O(\log D + \log\log_{F/n} n)$ becomes better the larger the global memory $F$ is. A limitation to solving connected components through independent node/edge contractions comes from the global memory bound. If the graph is already sparse, then the sparsification cannot make the graph any sparser, and hence we do not have an advantage in terms of global memory on the sparsified graph. In the case that $m = O(n)$ and the global memory is linear in $n$, the best we could hope for in the first round of contractions is to drop a constant fraction of the nodes. The low-level details for the reasons behind this can be extracted from the analysis of [2, 27, 24, 23]. Through the relative increase in global memory, the (remainder) graph size can be bounded by $n \cdot 2^{-2^i}$ in the $i$th round of contractions, which leads to an $\Omega(\log\log n)$ runtime.

In previous works, there is even more evidence towards sparse graphs being the hardest instances. Recently, it was shown that lower bound results from the LOCAL model of distributed message passing can be lifted to MPC under certain conditions [26, 24]. In the LOCAL model, almost all hardness results are obtained on trees or in high-girth graphs, implying lower bounds on forests with potentially many connected components [30, 5, 6, 8, 7, 11]. It was shown that a *component-stable* algorithm cannot solve a problem $\pi$ faster than in $O(\log T(n, \Delta))$, where $T(n, \Delta)$ is the complexity of $\pi$ in the LOCAL model[1] on a graph with $n$ nodes and maximum degree $\Delta$. Roughly speaking, an MPC algorithm is component-stable if the output on each node $u$ only depends on the size of the graph and the connected component of $u$ (see Definition 5.4 for more details [24]). While these methods do not yield unconditional hardness in the MPC model, we face similar difficulties in sparse graphs in the MPC model as in the message passing models.

**Rooted Forests and Applications to Locally Checkable Problems.** We believe that our technique to obtain connected components is of interest beyond solving the connectivity problem. For example, through minor adjustments to our technique, we obtain an algorithm that roots an (unrooted) input forest. Furthermore, we show that in a rooted tree, all *Locally Checkable Labeling (LCL)* problems can be solved very efficiently through a canonical algorithm. This generalizes to forests and gives an algorithm that can be executed on each connected component independently of the other components.

[1] The LOCAL algorithm is allowed to access shared randomness.

**Locally Checkable Labelings on Forests.** On the family of forests with component-wise maximum diameter $D$, all LCL problems can be solved deterministically in $O(\log D)$ time in the low-space MPC model with $O(n+m)$ global memory. Under the 1 vs. 2 cycles conjecture, this is optimal.

A range of central graph problems, in particular in the area of parallel and distributed computing, are locally checkable, where the correctness of the whole solution can be verified by checking the partial solution around the local neighborhood of each node. In particular, the class of LCL problems consists of problems with a finite set of outputs per node/edge and a finite set of locally feasible solutions (see Definition 6.2), and includes fundamental problems such as MIS, node/edge-coloring and the algorithmic Lovász Local Lemma (LLL). Our work shows that any LCL problem can be solved in $O(\log D)$ time and that the same runtime can be obtained for many problems that are not restricted to finite descriptions. We complement our results by showing that for LCL problems, this bound is tight under the 1 vs. 2 cycles conjecture.

In recent works, the complexity of LCLs in MPC was compared against *locality* [17, 4], where locality refers to the round complexity of solving an LCL in the LOCAL model, as a function of $n$. It was shown that all LCLs on trees can be solved exponentially faster in MPC as compared to LOCAL. As a consequence, all LCLs on trees can be solved in $O(\log n)$ rounds in the low-regime MPC model. We note that it is often the case that the diameter of a graph is small, potentially much smaller than the locality of a certain graph problem (which is *independent* of the diameter). Hence, our novel technique significantly improves on the state-of-the-art runtimes for various graph problems in a broad family of graphs.

**1.1 Our Contributions** Our main contribution is an algorithm that deterministically detects the connected components of a forest in time logarithmic on the maximum component-wise diameter; crucially, independent of the size $n$ of the input graph, whose dependence is inherently present in the techniques used in previous works. We also show that our approach is asymptotically optimal under the 1 vs. 2 cycles conjecture. Next, we present our results more formally.

THEOREM 1.1. (CONNECTED COMPONENTS) *Consider the family of forests. There is a deterministic low-space MPC algorithm to detect the connected components on this family of graphs. In particular, each node learns the maximum ID of its component. The algorithms runs in $O(\log D)$ rounds, where $D$ is the maximum diameter of any component. The algorithm requires $O(n+m)$ words of global memory, it is component-stable, and it does not need to know $D$. Under the 1 vs. 2 cycles conjecture, the runtime is asymptotically optimal.*

The techniques for Theorem 1.1 can be extended to also obtain a rooted forest, where each node also knows the ID of the corresponding root.

THEOREM 1.2. (ROOTING) *Consider the family of forests with component-wise maximum diameter $D$. There is a deterministic low-space MPC algorithm that roots the forest in $O(\log D)$ rounds using $O(n+m)$ words of global memory, and it is component-stable.*

The rooting of the input forest gives us a handle for easier algorithm design and memory allocation in low-space MPC. As a concrete example, our results yield an $O(\log D)$ algorithm for deterministically 2-coloring forests. Without going into technical details, this can be achieved through a rather simple algorithm, where each node decides its color based on the parity of its distance to the root, and only needs to keep one pointer in memory for the parity counting. In a sense, we outsource the tedious implementation details to the rooting algorithm in Theorem 1.2 and obtain a convenient tool for algorithm design.

More broadly, we show how to solve any LCL problem in rooted forests in $O(\log D)$ deterministic rounds. LCLs have gotten ample attention in various distributed models of computation, e.g., [17, 10, 12, 20, 21]. Roughly speaking, the family of LCLs is a subset of the problems for which we can check if a given solution is correct by inspecting the constant radius neighborhood of each node. (see Definition 6.2 for a formal definition of LCLs). Furthermore, we show that for any fixed $D \in \Omega(\log n)$ and $D \in n^{o(1)}$, there cannot exist an algorithm that solves all LCL problems in time $o(\log D)$ in the family of unrooted forests of diameter at most $D$. This holds even if $\text{poly}(n)$ global memory is allowed.

THEOREM 1.3. (LCLs ON TREES) *Consider an LCL problem $\Pi$ on forests and let $D$ be the component-wise maximum diameter. There is a deterministic low-space MPC algorithm that solves $\Pi$ in $O(\log D)$ rounds using $O(n+m)$ words of global memory. Under the 1 vs. 2 cycles conjecture, the runtime is asymptotically optimal.*

**1.2 Challenges and Techniques** A canonical approach to solve connected components on forests is to root each tree and identify each tree with the ID of the root. Also, examining the challenges in rooting demonstrates the challenges we face when identifying connected components. A natural approach to root a tree is to iteratively perform *rake* operations, i.e., pick all the leaves of the tree and each leaf picks the unique neighbor as its parent. This approach clearly roots a tree in $O(D)$ parallel rounds and furthermore, in the case of a forest, each tree performs its rooting process independently. If we ignore the memory considerations in the low-space MPC model, this process could be implemented in $O(\log D)$ rounds using the graph exponentiation technique, where, in $O(\log D)$ rounds, *every* node gathers their $D$-hop neighborhoods, i.e., the whole graph, to simulate the process fast. However, when we limit the global memory to $O(n+m)$, we get into trouble. A simulation through graph exponentiation requires that *all* nodes iteratively gather larger and larger neighborhoods *simultaneously*. With the strict memory bound, this implies that a node can only gather a *constant* radius neighborhood (and in non-constant degree graphs that we deal with even that is not possible!), which allows only for simulating a constant number of rake-iterations in one MPC round.

A hope towards a more efficient approach is to show that the amount of total memory *relative* to the nodes remaining in the graph increases as we rake the graph (similarly to previous work [2, 14, 24]). If one can reduce the size of the graph by a constant factor in each MPC round, then the available total memory increases by a constant factor per remaining node. Then, we can gather a slightly larger neighborhood per node in the next step of the simulation. However, even if we had this guarantee, the best we could hope for is a runtime that depends on $n$, since this approach relies on a progress measure that depends on shrinking the graph. Informally speaking, this observation says that we need to have a fundamentally different approach than gradually sparsifying the graph.

**Balanced Exponentiation.** One of our main technical contributions is to introduce a new method to gather a *part* of the neighborhood of each node that is balanced in the following sense. Suppose, for the sake of argument, that we have a rooted tree. Then, if a node $u$ has, say, $\gamma$ descendants, we ensure that $u$ will only gather $O(\gamma)$ nodes in the direction of the root, i.e., the direction opposing its descendants. Furthermore, it will also gather its $\gamma$ descendants, resulting in a memory demand of $O(\gamma)$ (for $u$). A crucial step in our analysis is to show that even if each node gathered their $\gamma$ descendants and $O(\gamma)$ nodes in the direction of the root, we do not create too much redundancy and we respect the linear total memory bound. A key technical challenge here is that there is no way for a node to know who are its descendants (because the input graph is unrooted). We show that, without an asymptotic loss in the runtime, we can deterministically determine which neighbor of $u$ is the *worst case* for a choice of a parent and gather the respective nodes slower.

**Progress Measure.** As mentioned above, to obtain a runtime independent of $n$, we need to avoid arguments that are based on the size of the graph getting smaller during the execution of our algorithm. The topology gathering through exponentiation can be seen as creating a virtual graph, where a virtual edge $\{u, v\}$ corresponds to the fact that $u$ knows how to reach $v$ and vise versa. The base of our progress measure is to aim to show that in this virtual graph, the diameter is reduced by a constant factor in each iteration. Unfortunately, having this type of guarantee seems too good to be true. Already on a path, it requires too much memory to create a virtual graph where the distances between all pairs of nodes are reduced. Our contribution is to show that this example is degenerate in the sense that either we can guarantee that the balanced exponentiation reduces the diameter or we can reduce it through a node-contraction type of operation.

**1.3 Further Related Work** In relation to our work, previous works have studied finding rooted spanning forests. In [15, 2], $O(\log D \cdot \log\log n)$ algorithms for rooting were given and the runtime was improved to $O(\log D + \log\log n)$ by [22].

Locally checkable problems have been intensively studied in the MPC model. Many classic algorithms from PRAM imply MPC algorithms with the same runtime, e.g., the MIS, maximal matching and coloring [34, 1]. The runtime of such simulations are typically polylogarithmic and, in MPC, the aim is to obtain something significantly faster. For MIS and maximal matching, there are $\widetilde{O}(\sqrt{\log \Delta} + \log\log\log n)$ time algorithms [27] and $(\Delta+1)$-node-coloring can be solved in $O(\log\log\log n)$ rounds, even deterministically [19, 24].

Many of the current state-of-the-art algorithms for locally checkable problems are (at least to some degree) based on distributed message-passing algorithms. The common design pattern is to design a message-passing algorithm, for example in the LOCAL model of distributed computing [33] where the output of each node is decided according to their $t$-hop neighborhood in $t$-rounds. These algorithms are then implemented faster in the MPC model through the graph exponentiation technique [31] that, in the ideal case, collects the $t$-hop ball around each node in $O(\log t)$-rounds. This framework was used to obtain an exponential speedup for many locally checkable problems in general, and in particular, it was used recently to show that all LCL problems on trees with $t$-round complexity in LOCAL can be solved in $O(\log t)$ MPC rounds [17, 4]. Our work broadens our understanding on the complexities of LCLs as a function of the diameter, which is somewhat orthogonal to previous works.

On a technical level, a related work gave a clever approach to encode the feasible outputs around each node into a constant sized *type* of the node [21]. Given a rooted tree, the type of a node $u$ (or its subtree) is determined through the set of possible outputs of its descendants. This encoding gives rise to an efficient convergecast protocol, where the root learns its type and effectively broadcasts a valid global solution to the rest of the tree. In a recent work, related techniques were used to implement a message passing algorithm for LCLs on trees using small messages [10]. In our work, we employ similar ideas to aggregate and broadcast information efficiently through the input tree.

**Lower Bounds.** While simulating LOCAL message passing algorithms in MPC has been fruitful in algorithm design, there is an inherent limitation to this approach. A naïve implementation results in a *component-stable* algorithm, where we can show that the simulation cannot be more than exponentially faster than the message passing algorithm [26, 24]. An algorithm is said to be *component-stable* if the output on node $v$ depends (deterministically) only on the topology, the input of the nodes, and the IDs of the nodes in the connected component of $v$. Furthermore, the output is allowed to depend on the number of nodes $n$, the maximum degree $\Delta$ of the input graph, and in case of randomized algorithms, the output can depend on shared randomness. It was shown that for component-stable algorithms and under the 1 vs. 2 cycles conjecture, $\Omega(\log t)$ rounds cannot be beaten if $t$ is a lower bound on the complexity of the given problem in the LOCAL model. We emphasize that our lower bounds also work for component-unstable algorithms (still relying on the 1 vs. 2 cycles conjecture).

## 2 Overview, Roadmap and Notation

Our formal results are presented in Sections 4 to 7. In Section 3, we present the core techniques of our algorithm. The formal version of this algorithm appears in Section 4.

**Section 4:** This section contains the most involved part of our work, i.e., an algorithm that lets every node of an input tree output the maximum identifier of the tree. On a tree $G$, our algorithm runs in $O(\log \hat{D})$ rounds and uses global memory $O((n+m)\cdot \hat{D}^3)$ where the parameter $\hat{D} \in [\mathrm{diam}(G), n^{\delta/8}]$ needs to be known to the algorithm. This sounds like a foolish approach, as this problem can be trivially solved in $O(1)$ rounds if we were really given a single tree as input. We still chose to present our result in this way, as our seemingly naïve algorithm is the core of our connected components algorithm that we present in Section 5.

**Section 5:** In fact, we show that our algorithm can be correctly extended to forests. Note that if every node knows the maximum identifier of its tree, we automatically solve the connected components problem. In this section, we also show how to remove the requirement of knowing $\hat{D}$ via doubly exponentially increasing guesses for $\hat{D}$. We also show that we can reduce the overall memory requirement to $O(n+m)$ by preprocessing the graph, that is, we spend additional $O(\log \hat{D})$ rounds to reduce the size of the graph by a factor $\hat{D}^3$. Lastly, we show that the runtime reduces to $O(\log \max_i\{D_i\})$ where $D_i$ is the diameter of the $i$-th component of the input graph. In this section, we also present the full proof of our connected components algorithm (Theorem 1.1) and our rooting algorithm (Theorem 1.2).

**Section 6:** In this section, we show a nice application of our rooting algorithm from Theorem 1.2. In particular, we show that any LCL problem can be solved in just $O(\log D)$ rounds, once each tree of the forest is rooted (Theorem 1.3).

Our approach has a dynamic programming flavor and we explain it for a single tree of the forest. We iteratively reduce the size of the tree, by compressing small subtrees into single nodes, and paths into single edges. While performing these compressions, we set additional constraints on the solution allowed on the nodes into which we compress subtrees, and on the edges that represent compressed paths. We maintain the invariant that, if we obtain a solution in the smaller tree, then it can be extended to the original one. We show that, by performing a constant number of compression steps, we obtain a tree that is comprised of a single node, where it

is straightforward to compute a solution. We then perform the same operations in the reverse order, in order to extend the solution to the whole tree. All of this is preceded by using Theorem 1.2 to compute a rooting of the tree/forest. The rooting helps, as with a given rooting it is significantly easier to identify the suitable subtrees to compress without breaking memory bounds.

**Section 7:** In this section, we show that the runtimes of our algorithms are tight, conditioned on the widely believed 1 vs. 2 cycles conjecture. Our aim is to use a reduction from the 1 vs. 2 cycles problem to solving connectivity on paths. In previous work [26], a reduction to connectivity on paths was introduced, but for technical reasons, it is not sufficient for our purposes. We require a guarantee that each path is of bounded diameter, which is not directly guaranteed by the previous work. Hence, we start by defining a problem on forests, called $D$-diameter $s$-$t$ path-connectivity, for which we can prove conditional hardness. By a reduction, we obtain a conditional lower bound of $\Omega(\log D)$ for the connected components problem.

We then define an LCL problem such that, given an algorithm for it, we can use it to solve the $D$-diameter $s$-$t$ path-connectivity problem. Hence, we obtain a lower bound of $\Omega(\log D)$ for the problem, implying that our generic LCL solver is also conditionally tight.

**2.1 Definitions and Notation** Given a graph $G = (V, E)$, we denote with $\Delta$ the maximum degree of $G$, with $n = |V|$ the number of nodes in $G$, and with $m = |E|$ the number of edges in $G$. We denote with $N_G(v)$ the neighbors of $v$, that is, the set $\{u \mid \{u, v\} \in E\}$. We denote with $\deg_G(v)$ the degree of a node $v$, that is, the number of neighbors of $v$ in $G$. If $G$ is clear from the context, we may omit $G$ and simply write $N(v)$ and $\deg(v)$. If $G$ is a directed graph, we denote with $\deg(v)$ the degree of $v$ in the undirected version of $G$, and with $\deg_{\text{in}}(v)$ and $\deg_{\text{out}}(v)$ its indegree and its outdegree, respectively. We define $\text{dist}_G(u, v)$ as the hop-distance between $u$ and $v$ in $G$. Again, we may omit $G$ if it is clear from the context. The *radius-$r$ neighborhood* of a node $v$ is the subgraph $G_r(v) = (V_r(v), E_r(v))$, where $V_r(v) = \{u \in V \ : \ \text{dist}(u, v) \leq r\}$, and $E_r(v) = \{(u, w) \in E \ : \ \text{dist}(v, u) \leq r \text{ and } \text{dist}(v, w) \leq r\}$. Also, we denote with $G^k$ the $k$-th power of $G$, that is, a graph containing the same nodes of $G$, where we connect two nodes $u$ and $v$ ($u \neq v$) if and only if they satisfy $\text{dist}_G(u, v) \leq k$. The *eccentricity* of a node $v$ in a graph $G$ is the maximum of $\{\text{dist}(u, v) \mid u \in V\}$.

## 3 The MAX-ID Problem: Overview and Techniques

In this section, we present the core techniques of our specialized algorithm for solving the MAX-ID problem, that is the core ingredient for solving connected components on forests (upper bound of Theorem 1.1). In the MAX-ID problem, one is given a connected tree with a unique identifier for each node, and all nodes must output the maximum identifier in the tree. We note that it is trivial to solve the problem in $O(1)$ MPC rounds using a broadcast tree; however, this approach does not extend to forests, and hence a more sophisticated solution is required. The purpose of this section is to present the high level ideas of an algorithm that solves MAX-ID and can also be extended to forests. Some lemma statements have been adapted to fit this (informal) version.

**Lemma 4.1** (Solving MAX-ID on trees)**.** *Consider the family of trees. There is a deterministic low-space MPC algorithm that solves MAX-ID on any graph $G$ of that graph family when given $\hat{D} \in [diam(G), n^{\delta/8}]$. The algorithms runs in $O(\log \hat{D})$ rounds, is component-stable[2], and requires $O(m \cdot \hat{D}^3)$ words of global memory.*

We begin with definitions that are essential not only to define our algorithm but also for proving its memory bounds. Let $v$ be a vertex of a tree $G$. For all nodes $u \in N(v)$, define

$$G_{v \to u} = \{w \in V(G) \mid u \text{ is contained in the shortest path from } v \text{ to } w\}$$

to be all nodes in the tree that are reachable from $v$ via $u$, including $u$. Also, let $G_{v \not\to u} \coloneqq V(G) \setminus G_{v \to u}$. For every $w \in G$, let $r_v(w)$ be the node $u \in N(v)$ satisfying that $w \in G_{v \to u}$, i.e., $r_v(w)$ is the neighbor of $v$ which is on the unique path from $v$ to $w$.

Definition (Light and heavy nodes) *Let $0 < \delta < 1$ be a constant. A node $v$ is* light against *a neighbor $u \in N(v)$ if $|G_{v \not\to u}| \leq n^{\delta/8}$. A node is* light *if it is light against at least one of its neighbors. Nodes that are not light are* heavy.

[2]By the formulation of Definition 4.1, any algorithm solving MAX-ID is component-stable by definition. This is discussed in detail in Section 5.4

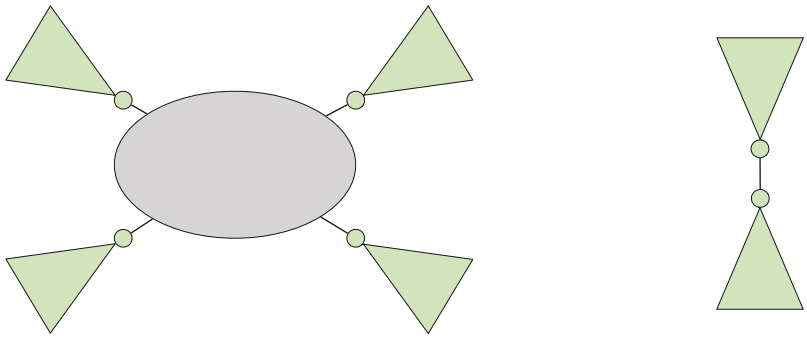

Figure 1: Light nodes are green and heavy nodes are gray.

If there are no heavy nodes, the graph is small and fits into the local memory of one machine.

Lemma (see Lemma 4.4) *Any tree with no heavy nodes contains at most* $2n^{\delta/2}$ *vertices.*

We prove that as soon as there is at least one heavy node, the graph has to look like the one depicted on the left hand side of Figure 1, that is, light subtrees that are attached to a connected component of heavy nodes. We exploit this structure in our algorithm.

**3.1 MAX-ID: The Algorithm** The high-level idea is to iteratively compress parts of the graph (without disconnecting it) such that the knowledge of the maximum identifier of the compressed parts is always kept within the resulting graph. We repeat this process until there remains only one node, that knows ID, the maximum identifier in the graph. Then, we backtrack the process by iteratively decompressing and broadcasting the knowledge about ID. Eventually, we are left with the original graph where all nodes know ID.

More in detail, our algorithm consists of $\ell = O(1)$ *phases* and the same number of *reversal phases*. During the phases, we first compress all light subtrees into single nodes (a procedure that we refer to as CompressLightSubTrees) and then replace all paths by a single edge (CompressPaths). We denote the resulting graphs by $G_0, G_1, \ldots, G_\ell$. The phases are followed by reversal phases, in which we undo all compression steps of the regular phases in reverse order to spread ID to the whole graph.

**Bounding the number of Phases.** Consider some phase $i$ and graph $G_i$ with heavy nodes that looks as illustrated in Figure 1 (left). If we remove all light subtrees from graph $G_i$, for the resulting graph $G_{i+1}$, it holds that every leaf (aka a formerly heavy node) corresponds to a distinct removed subtree of size at least $n^{\delta/8}$ (if the subtree was smaller the leaf would not be heavy). If we then contract all paths in $G_{i+1}$ into single edges, leaving no degree-2 nodes in $G_{i+1}$, it holds that at least half of the nodes in $G_{i+1}$ corresponds to a removed subtree. As each of these subtrees has $\geq n^{\delta/8}$ distinct nodes, we have removed a polynomial-in-$n$ fraction of nodes from $G_i$ to obtain $G_{i+1}$. Hence, we can only repeat the process a constant time until the graph becomes small.

**3.2 MAX-ID: Compressing Light Subtrees** For the sake of this high level overview, we focus on our most involved part, the procedure that compresses (maximal) light subtrees into the adjacent heavy node (CompressLightSubTrees). The difficulty is that nodes do not know whether they are light or heavy, and already one single exponentiation step in the "wrong" direction of the graph can ruin local and global memory bounds. However, there seems to be no way to obtain a runtime that is logarithmic in the diameter without exponentiation. Thus, we perform careful exponentiations that always ensure the memory bounds but at the same time make enough progress.

Consider a graph $G$ with $n$ nodes—starting from the second phase we will actually use this algorithm on graphs with fewer than $n$ nodes. At all times, every node $v$ has some set of nodes $S_v$ in its memory, which we initialize to $N(v)$. Set $S_v$ can be thought of as the node's view or knowledge. During the execution, $S_v$ grows, and if $|S_v| \geq 2n^{\delta/4}$, $v$ becomes full. Similarly to definitions $G_{v\to u}$ and $G_{v\not\to u}$, let us define the following. For a node $v$ and a node $u \in N(v)$, let $S_{v\to u} = S_v \cap G_{v\to u}$. Also, let $S_{v\not\to u} := S_v \setminus S_{v\to u}$.

All nodes in the graph have the property that they are either light or heavy. Initially, nodes themselves do not know whether they are light or heavy, since these properties depend on the topology of the graph. During

the algorithm each node is in one of the four states: active, happy, full, or sad. Initially, all nodes are active. A node $v$ becomes happy, if at some point during the execution, there exists $u \in N(v)$ such that $G_{v \not\to u} \subseteq S_v$ and $|G_{v \not\to u}| \leq n^{\delta/8}$. If a node, that is not full, realizes that it can never become happy (for example by having $|S_{v \to u}| > n^{\delta/8}$ for two different neighbors $u$), it becomes sad. Upon becoming happy, sad or full, nodes do not partake in the algorithm except for answering queries from active nodes. We call nodes unhappy if they are in some other state than happy (including state active). The goal is that all light nodes become happy. We will prove that the algorithm that we will provide satisfies the following lemma.

LEMMA 4.16. *After $O(\log \hat{D})$ iterations, all light nodes become happy, while heavy nodes always remain unhappy.*

Intuition for its correctness requires further details and is defered to the end of this section.

When comparing the definitions of happy and light, it is evident that when a node becomes happy, it knows that it is light. Similarly, a node becoming full or sad knows that it is heavy. At the end of the algorithm, happy nodes with a full or sad neighbor compress their whole subtree in that neighbor. A crucial challenge here is to ensure that these compressions are not conflicting as all such nodes execute these in parallel and without a global view.

**Exponentiation.** Recall the definition of $r_v(w)$ at the beginning of the section. For a node $v$ and any $X \subseteq N(v)$, define an exponentiation operation as

$$\mathsf{Exp}(X): \quad S_v \leftarrow \bigcup_{u \in X} \bigcup_{w \in S_{v \to u}} S_{w \not\to r_w(v)}.$$

We say that node $v$ *exponentiates in the direction of* $u \in N(v)$ if $v$ performs $\mathsf{Exp}(X)$ with $u \in X$.

The algorithm consists of $O(\log \hat{D})$ iterations, in each of which nodes perform a carefully designed graph exponentiation procedure. The aim is for light nodes $v$ to become happy by learning their subtrees, after which, (certain) light nodes compress into their unhappy neighbor.

**Failed Exponentiation Approaches.** If there were no memory constraints and every node could do a proper (uniform) exponentiation step in every iteration of the algorithm, i.e., execute $\mathsf{Exp}(N(v))$, after $O(\log \hat{D})$ iterations *all* nodes would learn the whole graph—a proper exponentiation step executed on all nodes halves the diameter—and the highest ID node could compress the whole graph into itself. However, uniform exponentiation would result in all nodes exceeding their local memory $O(n^\delta)$, and also significantly breaking the global memory requirement. Even if we were to steer the exponentiation procedure such that light nodes would learn a $D_v$ radius ball around them, where $D_v$ is the diameter of their light subtree, this would still break global memory. In fact, we cannot even do a single exponentiation step for all nodes in the graph without breaking memory bounds!

**Our Solution (Careful Exponentiation & Probing).** Hence, we let every node exponentiate in all but one direction, sparing the direction which currently looks most likely to be towards the heavy parts of the graph. Note that the knowledge of a node about the tree changes over time and in different iterations it may spare different directions. This step is further complicated as nodes neither know whether they are heavy or light nor do they know the size of their subtree, nor in which direction the heavy parts of the graph lie. Thus, in our algorithm, nodes perform a careful probing for the number of nodes into all directions to determine in which directions they can safely exponentiate without using too much memory. More formally, a node $v$ computes $B_{v \to u} = \sum_{w \in S_{v \to u}} |S_{w \not\to r_w(v)}|$ for every neighbor $u \in N(v)$ as an estimate for the number of nodes it may learn when exponentiating towards $u$. This estimate may be inaccurate and may contain a lot of doublecounting. The precise guarantees that this probing provides are technical and presented in Section 5.

We now reason (in a nutshell) why this algorithm meets the memory requirements and why we still make enough progress in order to make all light nodes happy in $O(\log \hat{D})$ iterations.

**Local and Global Memory Bounds.** If there were no memory limitation, we would already know that after $\ell = O(1)$ phases $G_\ell$ would consist of a single node $r$. For the sake of analysis, we assume a rooting of $G$ at $r$. We emphasize that fixing a rooting is only for analysis sake, and we do not assume that the tree is actually rooted beforehand.

Given the rooting at $r$, we define $T(v, r)$ as the subtree rooted at $v$ (including $v$ itself). Then, the following lemma is crucial to bound the memory. The lemma is standalone as it does not use any properties of $r$.

LEMMA 4.5. *Consider an $n$-node tree $T$ with diameter $D$ that is* rooted *at node $r$. Let $T(v, r)$ denote the subtree rooted at $v$ (including $v$) when $T$ is rooted at $r$. It holds that $\sum_{v \in V} |T(v, r)| \leq (D + 1) \cdot n$.*

*Proof.* Consider the unique path $P_{rv}$ from the root $r$ to a node $v$. Observe that node $v$ is only in the subtrees of the nodes in $P_{rv}$. Since $|P_{rv}| \leq D+1$, node $v$ is overcounted at most $D$ times, and $\sum_{v \in V} |T(v,r)| \leq (D+1) \cdot n$. ∎

The probing ensures that a node, if it exponentiates into a direction, essentially never learns more nodes than there are contained in its "rooted subtree" $T(v,r)$.

Lemma (see Lemma 4.17) *Let $v$ be any node with a parent $u$ (according to the hypothetical rooting at $r$). If in some iteration, node $v$ exponentiates in the direction of $u$, i.e., it performs* $\mathsf{Exp}(X)$ *with $u \in X$, the size of the resulting set $S_{v \to u}$ is bounded by $|T(v,r)| \cdot \hat{D}$.*

This is sufficient to sketch the global memory bound.

Lemma (see Lemma 4.18) *In* CompressLightSubTrees*, the global memory never exceeds $O(n \cdot \hat{D}^3)$.*

*Proof.* [Proof sketch] Assume that there is at least one heavy node and consider an arbitrary iteration $j$ of the algorithm. For node $v$ define the set $C_v \subseteq S_v$ as the set of nodes that $v$ has added to $S_v$ as a result of performing $\mathsf{Exp}$ in all iterations up to iteration $j$. Let $u$ be the parent of $v$ (according to the hypothetical rooting at $r$). For that $u$, let $C_{v \to u} \coloneqq C_v \cap G_{v \to u}$. We obtain

$$|C_v| \leq |C_{v \to u}| + \sum_{w \in N(v) \setminus u} |S_{v \to w}| \leq |T(v,r)| \cdot \hat{D} + |T(v,r)| = (1+\hat{D})|T(v,r)| \ .$$

The bound on $|C_{v \to u}|$ is obtained by applying the previous lemma for the last iteration where $v$ has exponentiated in the direction of $u$, and the bound on the sum is by the definition of $T(v,r)$.

We need to introduce the notation $C_v$, as in our actual algorithm, exponentiations are not symmetric. In order to ensure a *symmetric enough* view, nodes $v$ that add some vertex $u$ to their set $S_v$ also add themselves to the set $S_u$. Thus $C_v \neq S_v$. However, this results in at most a factor 2 increase in global memory. The total memory is then bounded by

$$\sum_{v \in V} |S_v| = \sum_{v \in V} 2|C_v| \leq \sum_{v \in V} 2(1+\hat{D})|T(v,r)| = O(n \cdot \hat{D}^2).$$

Here, the bound on $\sum_{v \in V} |T(v,r)|$ is due to Lemma 4.5. The additional $\hat{D}$ factor in the lemma statement is due to the fact that a node may learn about the same node $\hat{D}$ times in a single exponentiation step resulting in a local peak in global memory; details are given in the full proof. ∎

The bounds on local memory use that the probing ensures that we do not exponentiate into a direction if it would provide us with too many new nodes.

**Measure of Progress.** In order to show that all light nodes become happy, we prove that the distance between a light node and a leaf in its subtree decreases by a constant fraction in a constant number of rounds. Distance, in this case, can be measured via a virtual graph where there is an edge between two nodes $u$ and $v$ if $v \in S_u$ or $u \in S_v$. Our algorithm design ensures that light nodes always exponentiate in all but one direction. This is sufficient to show that each segment $x_1, \dots x_5$ of length 5 of a shortest path in the virtual graph $H$, shortens by at least one edge in each iteration. Intuitively, one can simply use that $x_3$ in such a segment either exponentiates in the direction of $x_1$ or $x_5$ and will hence add the respective node to its memory. The actual proof needs a more careful reasoning, e.g., as we cannot rely on $x_1$ being part of the memory of $x_2$, due to non homogeneous exponentiations in previous iterations.

## 4 The MAX-ID Problem

In this section, we give a specialized algorithm for the MAX-ID problem on trees, which will be the core ingredient for solving connected components on forests (upper bound of Theorem 1.1). Once having the algorithm for solving MAX-ID, one can extend it to work as a connected components algorithm. We defer this extension and its proofs to Section 5. We define the problem as follows.

Definition 4.1. (The MAX-ID problem) *Given a connected graph with a unique identifier for each node, all nodes output its maximum identifier.*

Lemma 4.1. (Solving MAX-ID on trees) *Consider the family of trees. There is a deterministic low-space MPC algorithm that solves MAX-ID on any graph $G$ of that graph family when given $\hat{D} \in [diam(G), n^{\delta/8}]$. The algorithms runs in $O(\log \hat{D})$ rounds, is component-stable[3], and requires $O(m \cdot \hat{D}^3)$ words of global memory.*

**4.1 Definition and Structural Results** We begin with structural properties of trees that are essential for proving our memory bounds. Also, the introduced notation plays a central role in each step of our algorithms. Let $v$ be a vertex of a tree $G$. For all nodes $u \in N(v)$, define

$$G_{v \to u} = \{w \in V(G) \mid u \text{ is contained in the shortest path from } v \text{ to } w\}$$

to be all nodes in the tree that are reachable from $v$ via $u$, including $u$. Also, let $G_{v \not\to u} \coloneqq V(G) \setminus G_{v \to u}$. For every $w \in G$ let $r_v(w)$ be $u \in N(v)$ such that $w \in G_{v \to u}$, i.e., $r_v(w)$ is the neighbor of $v$ which is on the unique path from $v$ to $w$.

Definition 4.2. (Light and heavy nodes) *Let $0 < \delta < 1$ be a constant. A node $v$ is* light against *a neighbor $u \in N(v)$ if $|G_{v \not\to u}| \leq n^{\delta/8}$. A node is* light *if it is light against at least one of its neighbors. When $v$ is light against $u$, let $T_{v,u}$ denote $G_{v \not\to u}$. Nodes that are not light are* heavy*.*

Observe that a light node $v$ can be light against multiple neighbors $u$ and hence, we need to use a subscript in the notation $T_{v,u}$. We emphasize that $v \in T_{v,u}$. Throughout most of our proofs we need to consider the cases that a (virtual) tree contains heavy nodes and the case that it only consists of light nodes separately. Both situations are depicted in Figure 1. We continue with proving structural properties for both cases. Any tree has light nodes as its leaves are light.

Lemma 4.2. *Consider a tree $G$ that contains a heavy node and let $v \in G$ be a light node against neighbor $u$. Then all nodes $x \in T_{v,u}$ are light. Moreover, any $x \in T_{v,y}$, $x \neq v$ is light against $r_x(v)$.*

*Proof.* The first part of the claim must holds since $G_{x \not\to r_x(v)} < G_{v \not\to u} \leq n^{\delta/8}$. The second part must hold, since otherwise, all nodes are light, contradicting the assumption that there exists a heavy node. ∎

Lemma 4.3. *For every tree $G$ with at least one heavy node, it holds that (i) heavy nodes induce a connected component, and that (ii) every light node is light against exactly one neighbor.*

*Proof.* For both parts, assume the opposite. Then there is a heavy node in $T_{v,u}$ for some light node $v$, which contradicts Lemma 4.2. ∎

Due to Lemma 4.3, we write $T_v$ instead of $T_{v,u}$ for a light node in a tree with (a) heavy node(s) and call $T_v$ the node's *subtree*.

Observation 4.1. *For any tree $G$ and any two adjacent nodes $u, v \in G$, we have $G = G_{v \not\to u} \cup G_{u \not\to v}$.*

*Proof.* Since $G_{v \to u} = G_{u \not\to v}$, we obtain $G_{u \not\to v} \cup G_{v \not\to u} = G_{v \to u} \cup G_{v \not\to u} = G$. ∎

Lemma 4.4. *Any tree with no heavy nodes contains at most $2n^{\delta/2}$ vertices.*

*Proof.* For the sake of analysis, let each node $v$ put one token on each incident edge $\{v, u\}$ where $v$ is light against $u$, i.e., $|G_{v \not\to u}| \leq n^{\delta/8}$. As all nodes are light the total number of tokens is at least as large as the number of nodes. Since the graph is a tree, at least one edge receives two tokens. Let $\{u, v\}$ be such an edge and observe that $G = G_{v \not\to u} \cup G_{u \not\to v}$ holds due to Observation 4.1. It holds that $|G| = |G_{v \not\to u} \cup G_{u \not\to v}| \leq 2n^{\delta/8}$, because $v$ is light against $u$ and $u$ is light against $v$. ∎

[3]By the formulation of Definition 4.1, any algorithm solving MAX-ID is component-stable by definition. This is discussed in detail in Section 5.4

The following lemma will be central to bounding the global memory of our algorithm. It considers a rooted tree, which we will only use for analysis; we do not assume a rooting is given as input.

LEMMA 4.5. *Consider an $n$-node tree $T$ with diameter $D$ that is* rooted *at node $r$. Let $T(v,r)$ denote the subtree rooted at $v$ (including $v$) when $T$ is rooted at $r$. It holds that $\sum_{v\in V} |T(v,r)| \leq (D+1)\cdot n$.*

*Proof.* Consider the unique path $P_{rv}$ from the root $r$ to a node $v$. Observe that node $v$ is only in the subtrees of the nodes in $P_{rv}$. Since $|P_{rv}| \leq D+1$, node $v$ is overcounted at most $D$ times, and $\sum_{v\in V} |T(v,r)| \leq (D+1)\cdot n$. □

**4.2 MAX-ID: The Algorithm** In this section, we present a MAX-ID algorithm for trees, which we refer to as MAX-ID-Solver. In our algorithm, every node of an input tree $G$ outputs the maximum identifier of the tree, which we denote by $\underline{\mathsf{ID}}$. We assume we are given $\hat{D} \in [\mathrm{diam}(G), n^{\delta/8}]$. The runtime of our algorithm is $O(\log \hat{D})$ and it requires $O(n \cdot \hat{D}^3)$ words of global memory.

The high-level idea is to iteratively compress parts of the graph (without disconnecting it) such that the knowledge of the maximum identifier of the compressed parts is always kept within the resulting graph. We repeat this process until there remains only one node, that knows $\underline{\mathsf{ID}}$. Then, we backtrack the process by iteratively decompressing and broadcasting the knowledge about $\underline{\mathsf{ID}}$. Eventually, we are left with the original graph where all nodes know $\underline{\mathsf{ID}}$.

As reasoned in Section 1 it is far from clear how to implement this simple outline with neither breaking the runtime nor the global memory bounds. From a high level point of view our algorithm consists of $O(1)$ **phases** and $O(1)$ **reversal phases**. During the phases, we first compress all light subtrees into single nodes (a procedure that we refer to as CompressLightSubTrees) and then replace all paths by a single edge (CompressPaths). In this section, we blackbox the properties of both procedures and prove that $O(1)$ phases are sufficient to reduce the graph to a single node (Lemma 4.11). By far the most technically involved part of our algorithm is the procedure CompressLightSubTrees, which we explain in detail in Section 4.3. The phases are followed by reversal phases, in which we undo all compression steps of the regular phases in reverse order to spread $\underline{\mathsf{ID}}$ to the whole graph.

Let us be more formal and define the compression/decompression steps. Throughout the algorithm, every node $v$ keeps track of a variable $\mathsf{id}_v$, which is initially set to be the identifier of $v$. The intuition behind variable $\mathsf{id}_v$ is that it represents the largest identified $v$ has "seen" so far. Let us define compressing and decompressing operations for node $v$ and any node set $X$. Note that decompressing $X$ from $v$ is only defined for $X, v$ such that $X$ was at some point compressed into $v$.

- Compress $X$ into $v$: set $\mathsf{id}_v \leftarrow \max_u\{\mathsf{id}_u \mid u \in X\}$ remove $X$ (and its incident edges) from the graph. For any edge $\{x,y\}$ with $x \in X$ and $v \neq y \notin X$ we introduce a new edge $\{v,y\}$.
- Decompress $X$ from $v$: set $\mathsf{id}_u \leftarrow \mathsf{id}_v$, $\forall u \in X$ and add $X$ (and its incident edges) back to the graph. Remove any edge $\{v,y\}$ that was added during the compression step of $X$ into $v$.

**Phases.** We initialize $G_0$ as the input graph. From $G_0$, we derive a sequence $G_1, G_2, \ldots, G_\ell$ of smaller trees until eventually, for some $\ell = O(1)$, it holds that $G_\ell = \{v\}$ for which $\mathsf{id}_v = \underline{\mathsf{ID}}$. The tree $G_{i+1}$ $(0 < i \leq \ell)$ is obtained from $G_i$ as follows: first compressing all light subtrees via CompressLightSubTrees$(G_i, \hat{D})$ and call the resulting tree $G'_i$, then $G_{i+1}$ is the result of compressing all paths of $G'_i$ into single edges via CompressPaths$(G'_i, \hat{D})$.

Throughout the sequence, we maintain the properties that compressions do not overlap, every $G_i$ is connected and non-empty, and that $\mathsf{id}_w = \underline{\mathsf{ID}}$ for *some* node $w \in G_i$.

**Reversal Phases.** From $G_\ell = \{v\}$, we derive a reversal sequence $G_{\ell-1}, G_{\ell-2}, \ldots, G_0$ such that any $G_i$ $(\ell > i \geq 0)$ has the same node and edge sets as $G_i$ during the regular phases, and $\mathsf{id}_w = \underline{\mathsf{ID}}$ for *every* node $w \in G_i$. The tree $G_{i-1}$ is obtained from $G_i$ as follows: first decompressing all paths via DecompressPaths$(G_i)$ and call the resulting tree $G'_{i-1}$, then $G_{i-1}$ is the result of decompressing all light subtrees via DecompressLightSubTrees$(G'_{i-1})$. Note that in reversal phase $i$ we only decompress paths and subtrees that were compressed during the regular phase $i$.

MAX-ID-Solver$(G, \hat{D})$

Initialize $G_0 \leftarrow G$

1. For $i = 0, \ldots, \ell - 1$ phases:
   (a) $G'_i = \mathsf{CompressLightSubTrees}(G_i, \hat{D})$
   *// If there are heavy nodes, all light nodes are compressed into the closest heavy node. Otherwise, all nodes are light and are compressed into a single node.*
   (b) $G_{i+1} = \mathsf{CompressPaths}(G'_i, \hat{D})$
   *// All paths are compressed into single edges.*
2. For $i = \ell - 1, \ldots, 0$ reversal phases:
   (a) $G'_i = \mathsf{DecompressPaths}(G_{i+1})$
   *// All paths that were compressed during Step 1(b) are decompressed.*
   (b) $G_i = \mathsf{DecompressLightSubTrees}(G'_i)$
   *// All light nodes that were compressed during Step 1(a) are decompressed from $v$.*

The correctness of MAX-ID-Solver is contained in the following lemma.

LEMMA 4.6. *There exists some $\ell$ such that*

1. *after $\ell$ phases, graph $G_\ell$ consists of exactly one node $v$ for which $\mathsf{id}_v = \underline{\mathsf{ID}}$.*
2. *after $\ell$ reversal phases, graph $G_0$ is the input graph and all nodes know $\underline{\mathsf{ID}}$.*

*Proof.* The proof is straightforward, given the thee essential lemmas (Lemmas 4.7 to 4.9) on the subroutines that we prove in the sections hereafter. Let us prove the two claims separately.

1. Consider graph $G_i$ at the start of any phase $i$. We first claim that $G_i$ never becomes empty during phase $i$, for which there are two cases: either $G_i$ contains heavy nodes, or all nodes in $G_i$ are light. In the case of the former: in Step 1(a), by Lemma 4.7, if there are heavy nodes in the graph, they are never compressed. In Step 1(b), by Lemma 4.8, all degree-2 nodes are compressed into single edges, leaving the graph non-empty. In the case of the latter, by Lemma 4.7, we are left with a single node. Observe that since any tree always contains light nodes (leaves are always light), the number of nodes decreases in every phase, and the first part of the claim 1 holds for some $\ell$. Since $G_\ell = \{v\}$ is a result of consecutive compression steps applied to the input graph $G_0$ without disconnecting it, by the definition of compression, it holds that $\mathsf{id}_v = \underline{\mathsf{ID}}$.
2. **Observation.** *Graph $G_i$ during reversal phases $i$ has the same node and edge sets as graph $G_i$ during phase $i$.*

   *Proof.* We prove the claim by induction. The base case holds since $G_\ell$ from Step 1 is given directly to Step 2 as input. Assume that the claim holds for reversal phase $i + 1$. By Lemma 4.9, all nodes that were compressed in phase $i$ during Step 1(a) (resp. (b)) can decompress themselves in reversal phase $i$ during Step 2(b) (resp. (a)), proving the claim. □

   Consider graph $G_\ell$ that consists of a single node $v$ for which $\mathsf{id}_v = \underline{\mathsf{ID}}$ by Lemma 4.6. Since graph $G_0$ after $\ell$ reversal phases (which is the input graph by the observation above) is a result of consecutive decompression steps applied to $G_\ell$, by the definition of decompression, it holds that $\mathsf{id}_u = \underline{\mathsf{ID}}$ for all $u \in G_0$.

□

LEMMA 4.7. (CompressLightSubTrees) *Let $G$ be a tree and $\hat{D} \in [diam(G), n^{\delta/8}]$. If $G$ contains a heavy node, then $\mathsf{CompressLightSubTrees}(G, \hat{D})$ returns a tree in which all light nodes of $G$ are compressed into the closest heavy node. If $G$ does not contain any heavy nodes, all nodes are compressed into a single node. The algorithm runs in $O(\log \hat{D})$ low-space MPC rounds using $O(n \cdot \hat{D}^3)$ words of global memory.*

LEMMA 4.8. (CompressPaths) *For any tree $G$ and $\hat{D} \in [diam(G), n^{\delta/8}]$,* CompressPaths$(G, \hat{D})$ *returns the graph that is obtained from $G$ by replacing all paths of $G$ with a single edge. The algorithm runs in $O(\log \hat{D})$ low-space MPC rounds using $O(n \cdot \hat{D}^2)$ words of global memory.*

LEMMA 4.9. (DecompressPaths,DecompressLightSubTrees) *All nodes that were compressed by* CompressLightSubTrees *and* CompressPaths *can be decompressed by* DecompressLightSubTrees *and* DecompressPaths*, respectfully. The algorithms run in $O(1)$ low-space MPC rounds using $O(n)$ words of global memory.*

We will now show that the number of phases of (and therefore reversal phases) is bounded by $O(1)$. In particular, we want to prove that after $\ell = O(1)$ phases, graph $G_\ell$ consists of exactly one node. After a clever observation in Lemma 4.10, we will prove the claim in Lemma 4.11.

LEMMA 4.10. *If $|G_{i+1}| \geq 2$, all nodes in $G_{i+1}$ were heavy in $G_i$. Moreover, for every leaf node $w \in G_{i+1}$ it holds that $\geq n^{\delta/8}$ light nodes were compressed into $w$ during phase $i$.*

*Proof.* Since $|G_{i+1}| \geq 2$ (and not $|G_{i+1}| = 1$), by Lemma 4.7, there must have been heavy nodes in $G_i$. Since all light nodes were compressed in phase $i$, all nodes in $G_{i+1}$ were heavy in $G_i$. Observe that even though $w$ is a leaf in phase $i+1$, it was not a leaf node in phase $i$, since leaf nodes are light by definition. Let $u$ be the unique neighbor of $w$ in $G_{i+1}$. We must show that $|G_{w \not\to u}| > n^{\delta/8}$ and that $G_{w \not\to u} \setminus w$ was compressed into $w$ during phase $i$. It must be that $|G_{w \not\to u}| > n^{\delta/8}$, since otherwise, $w$ would have been light against $u$ in phase $i$. Nodes $G_{w \not\to u} \setminus w$ were compressed into $w$ during phase $i$ by Lemma 4.7, since $w$ was their closest heavy node (due to the graph being a tree). ∎

LEMMA 4.11. *After $\ell = O(1)$ phases, graph $G_\ell$ consists of exactly one node.*

*Proof.* Consider graph $G_i$ at the beginning of some phase $i$. If there are no heavy nodes in $G_i$, this is the last phase of the algorithm by Lemma 4.7. If there is exactly one heavy node in $G_i$, we are also done by Lemma 4.7. What remains to be proven is that if there are at least two heavy nodes in the graph, we reduce the size of the graph by a polynomial factor in $n$.

Assume that there are at least 2 heavy nodes in graph $G_i$, and let us analyze what happens. In Step 1(a), all light nodes are compressed into the closest heavy node by Lemma 4.7. In Step 1(b), all paths are compressed into single edges by Lemma 4.8, leaving no degree-2 nodes in the graph (compressing paths never creates new degree-2 nodes). Consider graph $G_{i+1}$, which by Lemma 4.7 consists of the nodes that were heavy in $G_i$. By Lemma 4.10 it also holds that during phase $i$, at least $n^{\delta/8}$ light nodes were compressed into every leaf node $w$ of graph $G_{i+1}$. It holds that

$$n_i \geq n_{i+1} + |\{w \in G_{i+1} \mid \deg_{G_{i+1}}(w) = 1\}| \cdot n^{\delta/8} > n_{i+1} + n^{\delta/8} \cdot n_{i+1}/2 = n_{i+1}(1 + n^{\delta/8}/2)$$

and $n_{i+1} < n_i/(1 + n^{\delta/8}/2) < 2n_i/n^{\delta/8}$ .

The first strict inequality stems from the fact that there are no degree-2 nodes left after phase $i$, and hence the number of leaf nodes in $G_{i+1}$ is strictly larger that $n_{i+1}/2$. The proof is complete, as we have shown that if graph $G_i$ contains at least 2 heavy nodes, $G_{i+1}$ is smaller than $G_i$ by a factor of $\Theta(n^{\delta/8})$. ∎

The outline for the rest of this section is as follows. The procedure CompressLightSubTrees and the proof of Lemma 4.7 are presented in Section 4.3. This is the most technically involved part of our algorithm. The procedure CompressPaths and the proof of Lemma 4.8 are presented in Section 4.4. The procedures DecompressPaths and DecompressLightSubTrees and the proof of Lemma 4.9 are presented in Section 4.5. In Appendix A, we show technical details how MAX-ID-Solver can be implemented in the low-space MPC model.

**4.3 MAX-ID: Single Phase** (CompressLightSubTrees) In this section, we focus on a single execution of CompressLightSubTrees$(G, \hat{D})$ on a graph $G$ and prove Lemma 4.7. With out loss of generality, we assume there are $n$ nodes in the graph—starting from the second phase of MAX-ID-Solver we will actually use this algorithm on graphs with fewer than $n$ nodes.

At all times, every nodes $v$ has some set of nodes $S_v$ in its memory, which we initialize to $N(v)$. Set $S_v$ can be thought of as the node's view or knowledge. During the execution, $S_v$ grows, and if $|S_v| \geq 2n^{\delta/4}$, $v$ becomes

full. Similarly to definitions $G_{v\to u}$ and $G_{v\not\to u}$, let us define the following. For a node $v$ and a node $u \in N(v)$, let $S_{v\to u} = S_v \cap G_{v\to u}$. Also, let $S_{v\not\to u} \coloneqq S_v \setminus S_{v\to u}$. Recall the definition of $r_v(w)$: for every $w \in G$ let $r_v(w)$ be $u \in N(v)$ such that $w \in G_{v\to u}$.

All nodes in the graph have the property that they are either light or heavy (see Definition 4.2). Initially, nodes themselves do not know whether they are light or heavy, since these properties depend on the topology of the graph. During the algorithm each node is in one of the four states: active, happy, full, or sad. Initially, all nodes are active. A node $v$ becomes happy, if at some point during the execution, there exists $u \in N(v)$ such that such that $G_{v\not\to u} \subseteq S_v$ and $|G_{v\not\to u}| \leq n^{\delta/8}$. In that case, we say that node $v$ is *happy against* $u$. If a node, that is not full, realizes that it can never become happy (for example by having $|S_{v\to u}| > n^{\delta/8}$ for two different neighbors $u$), it becomes sad. Upon becoming happy, sad or full, nodes do not partake in the algorithm except for answering queries from active nodes. We call nodes unhappy if they are in some other state than happy (including state active). The goal is that all light nodes eventually become happy, and heavy nodes always remain unhappy. When comparing the definitions of happy and light, it is evident that when a node becomes happy, it knows that it is light. Similarly, a node becoming full or sad knows that it is heavy.

For a node $v$ and any $X \subseteq N(v)$, define an exponentiation operation as

$$\mathsf{Exp}(X): \quad S_v \leftarrow \bigcup_{u\in X} \bigcup_{w\in S_{v\to u}} S_{w\not\to r_w(v)}.$$

We say that a node $v$ *exponentiates towards (or in the direction of)* $u$ if $u \in N(v)$ and $v$ performs $\mathsf{Exp}(X)$ with $u \in X$.

**High level overview of** CompressLightSubTrees**.** The algorithm consists of $O(\log \hat{D})$ iterations, in each of which nodes perform a carefully designed graph exponentiation procedure. The aim is for light nodes $v$ to become happy by learning their subtrees $T_v$, after which, (certain) light nodes compress $T_v$ into their unhappy neighbor. If there were no memory constraints and every node could do a proper (uniform) exponentiation step in every iteration of the algorithm, i.e., execute $\mathsf{Exp}(N(v))$, after $O(\log \hat{D})$ iterations *all* nodes would learn the whole graph—a proper exponentiation step executed on all nodes halves the diameter—and the highest ID node could compress the whole graph into itself. However, uniform exponentiation would result in all nodes exceeding their local memory $O(n^\delta)$, and also significantly breaking the global memory requirement. Even if we were to steer the exponentiation procedure such that light nodes would learn a $D_{T_v}$ radius ball around them, where $D_{T_v}$ is the diameter of their light subtree, this would still break global memory. In fact, we cannot even do a single exponentiation step for all nodes in the graph without breaking memory bounds! Hence, we need to steer the exponentiation with some even more stronger invariant in order to abide by the global memory constraint.

Observation 4.2. *If every light node $v$ keeps $O(|T_{v,u}|)$ nodes in its local memory for some (possibly unique) neighbor $u$ it is light against, this does not violate local memory $O(n^\delta)$ nor global memory $O(n \cdot \hat{D})$.*

*Proof.* If there is a heavy node in the graph, $|T_{v,u}|$ is unique by Lemma 4.3. The claim follows by considering a hypothetical rooting of the tree at some heavy node and applying Lemma 4.5. Otherwise, the claim holds trivially because the graph is of size $\leq 2n^{\delta/8}$ by Lemma 4.4. ∎

Inspired by the observation above, we aim to steer the exponentiation such that it is performed in a balanced way, where a node learns roughly the same number of nodes in each direction (or sees only leaves in one direction). In fact, we do not want to exponentiate in a direction if that exponentiation step would provide us with $\gg |T_v|$ nodes. This step is further complicated as nodes neither know whether they are heavy or light nor do they know the size of their subtree. In our algorithm that is presented below we perform a careful probing for the number of nodes into all directions to determine in which directions we can safely exponentiate without using too much memory. In the probing procedure ProbeDirections, a node $v$ computes $B_{v\to u} = \sum_{w\in S_{v\to u}} |S_{w\not\to r_w(v)}|$ for every neighbor $u \in N(v)$ as an estimate for the number of nodes it may learn when exponentiating towards $u$. This estimate may be very inaccurate and may contain a lot of doublecounting. In Section 4.3.3, we present the full procedure and prove the following lemma.

Lemma 4.12. (ProbeDirections) *Consider an arbitrary iteration of algorithm* CompressLightSubTrees*. Then algorithm* ProbeDirections$(\hat{D})$ *returns:*

*(i)* $\mathsf{fullDirs} \subseteq N(v)$ *such that if we were to exponentiate in all directions, we would obtain* $|S_{v\to u'}| > n^{\delta/8}$ *for all* $u' \in \mathsf{fullDirs}$ *and* $|S_{v\to u'}| \le n^{\delta/8} \cdot \hat{D}$ *for all* $u' \in N(v) \setminus \mathsf{fullDirs}$.

*(ii)* $\mathsf{largestDir} \in N(v)$ *(returned if* $\mathsf{fullDirs} = \emptyset$*) such that if we were to exponentiate in all directions, we would obtain* $|S_{v\to \mathsf{largestDir}}| \ge |S_{v\to u'}|$ *for all* $u' \in N(v)$ *and* $|S_{v\to \mathsf{largestDir}}| \le n^{\delta/8} \cdot \hat{D}$.

ProbeDirections *can be implemented in* $O(1)$ *low-space MPC rounds, using* $O(n \cdot \hat{D}^3)$ *global memory. It does not alter the state of* $S_v$ *for any node* $v$ *in the execution of* CompressLightSubTrees.

The main difficulty of CompressLightSubTrees lies in ensuring the global and local memory constraints (Lemmas 4.18 and 4.19) that prevent us from blindly exponentiating in all directions, while at the same time ensuring enough progress for light nodes such that every light node becomes happy by the end of the algorithm (Lemma 4.16).

CompressLightSubTrees$(G_i, \hat{D})$

All nodes are active. Initialize $S_v \leftarrow N(v)$. If $|S_v| > n^{\delta/8} + 1$, $v$ becomes sad.

1. For $O(\log \hat{D})$ iterations:
   (a) $\mathsf{fullDirs}, \mathsf{largestDir} \leftarrow \mathsf{ProbeDirections}(\hat{D})$
   // The properties of $\mathsf{ProbeDirections}(\hat{D})$ are formally stated in Lemma 4.12. Informally, $\mathsf{fullDirs} \subseteq N(v)$ contains directions with $> n^{\delta/8}$ nodes, and largestDir contains the direction with the largest number of nodes if $\mathsf{fullDirs} = \emptyset$.
   (b) If $|\mathsf{fullDirs}| \ge 2$, $v$ becomes sad.
   (c) If $|\mathsf{fullDirs}| = 1$:
      i. Perform $\mathsf{Exp}(N(v) \setminus \mathsf{fullDirs})$
   (d) If $|\mathsf{fullDirs}| = 0$:
      i. Perform $\mathsf{Exp}(N(v) \setminus \mathsf{largestDir})$
      ii. If $v$ is in $S_w$ for some $w$, add $w$ to $S_v$ // ensure symmetric view
   (e) Node $v$ asks nodes $w \in S_v$ whether or not they are happy against $r_w(v)$, and if so, what is the size of subtree $T_{w,r_w(v)}$. Node $v$ can locally compute if it can become happy by learning subtrees $T_{w,r_w(v)}$. If $v$ can, it asks for them and becomes happy.

   // After Step 1, all light nodes are happy, and all heavy nodes are unhappy (Lemma 4.16)
2. Happy nodes $v$ with an unhappy neighbor $u$ compress $S_{v\not\to u} = G_{v\not\to u}$ into $u$.
3. Nodes $v$ that are happy against $u$ such that $u$ is happy against $v$ update $S_v \leftarrow S_v \cup S_u$ and compress $S_v$ into the highest ID node in $S_v$.

In Section 4.3.1, we discuss the measure of progress and correctness, with the final correctness proof of Lemma 4.7. In Section 4.3.2, we discuss local and global memory bounds, with the final memory proofs of Lemma 4.7. The MPC implementation is deferred to Appendix A.

**4.3.1 Measure of Progress and Correctness** We begin by proving the measure of progress and correctness, which will give us the means to analyze the memory requirements as if the tree was rooted.

LEMMA 4.13. *Let* $v$ *be a node that is light against neighbor* $u$. *If in some iteration of* CompressLightSubTrees, $v$ *exponentiates in the direction of* $u$, *i.e., it performs* $\mathsf{Exp}(X)$ *with* $u \in X$, *the size of the resulting set* $S_{v\to u}$ *is bounded by* $|T_v|$.

*Proof.* Consider an arbitrary iteration of the algorithm. If $u \in \mathsf{fullDirs}$, we do not exponentiate towards $u$, so there is nothing to prove. If $u \notin \mathsf{fullDirs}$, but $\mathsf{fullDirs} \neq \emptyset$, there is some $w \neq u$ such that, by the Probing Lemma 4.12, $|G_{v \to w}| > n^{\delta/8}$, which is a contradiction to $v$ being light against $u$.

Hence, consider the case that $\mathsf{fullDirs} = \emptyset$. If $\mathsf{largestDir} = u$, we do not exponentiate towards $u$ and there is nothing to prove. If $\mathsf{largestDir} \neq u$, then we exponentiate towards $u$ and by Lemma 4.12 $(ii)$, we have $|S_{v \to u}| \leq |G_{v \to \mathsf{largestDir}}| \leq |T_v|$. ∎

LEMMA 4.14. *In any iteration of* CompressLightSubTrees*, a light node neither becomes* full *nor* sad*.*

*Proof.* Node $v$ never becomes full due to initialization $S_v \leftarrow N(v)$, since for a light node it must hold that $|N(v)| \leq |T_v| + 1 \leq n^{\delta/8} + 1 < 2n^{\delta/4}$. During execution, $S_v$ grows only in Steps 1(c)–(e). During (c), it must be that $\mathsf{fullDirs} = u$, since otherwise it would imply that $|T_v| > n^{\delta/8}$. Hence, as a result of (c), $v$ cannot become full. During (d)i, if $\mathsf{Exp}(X)$ with $u \notin X$, it holds that $X \subset T_v$ and $v$ cannot become full. Otherwise if $u \in X$, by Lemma 4.13, $v$ cannot become full. Node $v$ cannot become full even when performing Step 1(d)ii, since a hypothetical exponentiation step in the direction of $u$ would yield a set that is bounded by $n^{\delta/8} \cdot \hat{D} < 2n^{\delta/4}$ (fullDirs is empty and $u$ is largestDir). During (e), node $v$ becomes happy against $u$ and hence $|S_{v \not\to u}| \leq n^{\delta/8}$. In the worst case, $|S_{v \to u}| < n^{\delta/8} \cdot \hat{D} \leq n^{\delta/4}$. Hence, as a result of (e), $v$ cannot become full.

A node can become sad only if its degree is too large, or in Step 1(b). A light node $v$ never becomes sad since it must hold that $|N(v)| \leq |T_v| + 1 \leq n^{\delta/8} + 1$, and $v$ cannot have two or more neighbors $u$ with $G_{v \to u} > n^{\delta/8}$ (one neighbor would have to be in $T_v$, implying that $|T_v| > n^{\delta/8}$). ∎

For the proofs of the next two lemmas, let $G = (V, E)$ be the input graph, and consider graph $G' = (V', E')$ such that $V' = V$ and $E' = E \cup \{\ \{v, w\} \mid v, w \in V \text{and } w \in S_v \text{ or } v \in S_w\}$.

LEMMA 4.15. (MEASURE OF PROGRESS) *At the start of any iteration $j$, consider a light (but still* active*) node $v$, and the longest shortest path $P^j_{vw}$ in $G'$ between $v$ and an a leaf node $w \in T_v$. If $|P^j_{vw}| \geq 4$ holds, then holds that $|P^{j+1}_{vw}| \leq \lceil 3/4 \cdot |P^j_{vw}| \rceil$ holds.*

*Proof.* Consider any subpath $P_{x_1 x_5} = \{x_1, x_2, x_3, x_4, x_5\} \subseteq P^j_{vw}$ of length 4. For $1 \leq i \leq 5$, let $S_{x_i}$ ($S'_{x_i}$) be the memory of node $i$ at the start (end) of iteration $j$. Note that all nodes on the path are light. By Lemma 4.14, nodes in $P_{x_1 x_5}$ never get full nor sad, and hence always exponentiate in all but one direction (either fullDirs or largestDir).

**Claim.** For $1 \leq i < 5$ it holds that $x_{i+1} \in S_{x_i}$.

*Proof.* Since edge $\{x_i, x_{i+1}\}$ exists in $G'$, it must be either that either $x_{i+1} \in S_{x_i}$ or $x_i \in S_{x_{i+1}}$. In the first case the claim holds, so consider the latter. Since $x_i$ is light, $r_{x_i}(x_{i+1})$ has never been in fullDirs for $x_i$. Hence, whenever $x_{i+1}$ had added $x_i$ to $S_{x_{i+1}}$, either $x_i$ added $x_{i+1}$ to $S_{x_i}$ via exponentiation, or via Step 1(d)ii. ∎

We continue with proving that the path shortens. It is sufficient to prove that for some $i, j \in [1, 5]$, $i \neq j$, it holds that $x_i \in S'_j \setminus S_j$, as this shortens the path between $x_1$ and $x_5$ by one edge.

1. If $x_2 \notin S_{x_3}$: Since there is an edge in $G'$ such that $x_2 \notin S_{x_3}$, it means that $x_2$ added $x_3$ to $S_{x_2}$ during some iteration $j$, and since $x_3$ did not add $x_2$ in Step 1(d)ii of iteration $j$, direction $r_{x_3}(x_2)$ must have been in fullDirs for $x_3$. Hence, $x_3$ will exponentiate in all directions besides $r_{x_3}(x_2)$. In particular, as $r_{x_3}(x_4) \neq r_{x_3}(x_2)$, $x_3$ will exponentiate towards $x_4$. As $x_5 \in S_{x_4}$, we obtain $x_5 \in S'_{x_3} \setminus S_{x_3}$. This creates an edge between $x_3$ and $x_5$ in $G'$ and shortens the path from 4 to 3, i.e., by a factor 3/4.

2. If $x_2 \in S_{x_3}$: Assume that $x_1 \in S_{x_2}$. Since nodes in $P_{x_1 x_5}$ exponentiate in all but one direction, node $x_3$ will exponentiate either towards $x_2$ or $x_4$ (it must be that $x_4 \in S_{x_3}$ by the claim above). If $x_3$ exponentiates towards $x_4$, we obtain $x_5 \in S'_{x_3} \setminus S_{x_3}$ as $x_5 \in S_{x_4}$. If $x_3$ exponentiates towards $x_4$, we obtain $x_1 \in S'_{x_3} \setminus S_{x_3}$ as $x_1 \in S_{x_2}$. If $x_1 \notin S_{x_2}$, we can apply the analysis of 1. for node $x_2$.

∎

LEMMA 4.16. *After $O(\log \hat{D})$ iterations, all light nodes become* happy*, while heavy nodes always remain* unhappy*.*

*Proof.* Let us adopt the notation of the proof of Lemma 4.15. Since Lemma 4.15 holds for any light node $v$, after $j = O(\log \hat{D})$ iterations it must holds that $|P^j_{vw}| \leq 3$ because $\hat{D} \in [\mathrm{diam}(G), n^{\delta/8}]$. Let the resulting path be $P = \{x_1, x_2, x_3, x_4\}$, where $x_4$ is a leaf node. It must be the case that if $x_2$ learns $S_{x_3 \not\mapsto x_2}$ for all possible nodes $x_3$, node $x_2$ becomes happy. Hence, in Step 1(e), $P$ shortens by one. Eventually, after two iterations, $P$ is of length one, and $x_1$ becomes happy.

For the second part of the claim it is sufficient to show that heavy nodes never become happy. Recall that heavy nodes are defined as nodes that are not light. Hence, for a heavy node $v$, there does not exist a neighbor $u \in N(v)$ such that $|G_{v \not\mapsto u}| \leq n^{\delta/8}$. This implies that during the algorithm, it is not possible for $|S_{v \not\mapsto u}| = |G_{v \not\mapsto u}| \leq n^{\delta/8}$ for any $u \in N(v)$. Hence, heavy nodes never become happy. ∎

*Proof.* [Proof of Lemma 4.7 (Correctness)] By Lemma 4.16, we know that after $O(\log \hat{D})$ iterations all light nodes of $G$ become happy, while all heavy nodes always remain unhappy. In order to prove the correctness of Lemma 4.7, we need to show that all light trees are compressed into the closest heavy node, if a heavy node exists, and that the whole tree is compressed into a single node if there are no heavy nodes. We consider both cases separately. Also consult Figure 1 for an illustration of both cases.

**Case 1 (there are heavy nodes).** Consider a light node $v$. As there are heavy nodes, Lemma 4.3 implies that there is a unique neighbor $u \in N(u)$ against which $v$ is light. Let $T_v = G_{v \not\mapsto u}$. Now, by Lemma 4.16, $v$ is happy at the end of the algorithm, i.e., there is a neighbor $u' \in N(v)$ for which $S_{v \not\mapsto u'} = G_{v \not\mapsto u'}$ and $|G_{v \not\mapsto u'}| \leq n^{\delta/8}$. The latter condition says that $v$ is light against $u'$ and due to the earlier discussion we deduce that $u = u'$ and $T_v = G_{v \not\mapsto u} = S_{v \not\mapsto u} \subseteq S_v$ holds. In summary, for every light node $v$, the tree $T_v$ (that does not depend on the algorithm) is contained in $S_v$. By Lemma 4.3, heavy nodes induce a single connected component, and hence every light node $v$ is contained in the subtree of some light node $v'$ that has a heavy neighbor $u'$. Since we are in a tree, $u'$ is the closest heavy node for $v'$, and in particular, for all light nodes $v \in T'_v$. By Lemma 4.16, $v'$ is happy at the end of the algorithm, and $u'$ remains unhappy. Performing Step 2 fulfills the first claim of Lemma 4.7. Step 3 is never performed, since there are no happy nodes left in the graph.

**Case 2 (all nodes of $G$ are light).** Step 2 of CompressLightSubTrees is never performed, since all (light) nodes are happy due to Lemma 4.16. For the sake of analysis, let each node $v$ put one token on each incident edge $\{v, u\}$ for which $G_{v \not\mapsto u} \subseteq S_v$ holds. As all (light) nodes are happy, i.e., there is a neighbor $u$ such that $G_{v \not\mapsto u} \subseteq S_v$ holds, the total number of tokens is at least as large as the number of nodes. Since the graph is a tree, at least one edge receives two tokens. Let $\{u, v\}$ be such an edge and observe that $G_{v \not\mapsto u} \subseteq S_v$ and $G_{u \not\mapsto v} \subseteq S_u$. Due to Observation 4.1, $G_{v \not\mapsto u} \cup G_{u \not\mapsto v} = G$ and after Step 3 of CompressLightSubTrees both nodes have the complete tree in their memory and both nodes trigger a compression of the whole tree into the largest ID node.

The edge $\{u, v\}$ with the above properties is not unique, but after Step 3, the endpoints of any edge having these properties yield the exact same compression. ∎

**4.3.2 Local and Global Memory Bounds** The most difficult part is proving the memory bounds when there are heavy nodes. If there were no memory limitation, Lemmas 4.6 and 4.11 (building up on versions of Lemmas 4.7 to 4.9 without memory limitations) already imply that after $O(1)$ phases of MAX-ID-Solver, there is exactly one node left in the graph. Denote this node by $r$. Node $r$ has never been compressed by definition. For the sake of analysis, we assume a rooting of $G$ at $r$. We emphasize that fixing a rooting is only for analysis sake, and we do not assume that the tree is actually rooted beforehand. We define $T(v, r)$ as the subtree rooted at $v$ (including $v$ itself), as if tree $G$ was rooted at $r$.

Observation 4.3. *Consider tree $G$ with at least one heavy node during an arbitrary iteration of* CompressLightSubTrees. *For every light node $v$, it holds that $T(v, r) = T_v$, and every heavy node $u$ has a unique subtree $T(u, r)$.*

Recall that the definition of $T_v$ for a light node $v$ was independent from any algorithmic treatment. Still, it holds that $T_v$ equals $T(v, r)$ (that depends on our algorithm as the node $r$ depends on it). The next lemma states that a node only exponentiates into the direction of root $r$ if it is safe to do so in terms of memory constraints. In spirit, it is very similar to Lemma 4.13, with the slight difference that it applies to all nodes, and we prove the claim using a hypothetical rooting of the tree.

Lemma 4.17. *Let $v$ be any node with a parent $u$ (according to the hypothetical rooting at $r$). If in some iteration of* CompressLightSubTrees *when there are heavy nodes, node $v$ exponentiates in the direction of $u$, i.e., it performs* Exp($X$) *with $u \in X$, the size of the resulting set $S_{v \to u}$ is bounded by $|T(v, r)| \cdot \hat{D}$.*

*Proof.* If $v$ performs $\mathsf{Exp}(X)$ such that $u \in X$, there must exists either $w = \mathsf{fullDirs}$ or $w = \mathsf{largestDir}$ such that $w \in T(v,r)$. If $w = \mathsf{fullDirs}$, it implies that $v$ is heavy, and by Lemma 4.12 $(i)$, we have that $|S_{v\to u}| \leq n^{\delta/8} \cdot \hat{D} < |T(v,r)| \cdot \hat{D}$. If $w = \mathsf{largestDir}$, by Lemma 4.12 $(ii)$, we have that $|S_{v\to u}| \leq |G_{v\to w}| \leq |T(v,r)|$. ∎

Observation 4.4. *When node $v$ performs an exponentiation step, multiple nodes $w$ can send the same node to $v$, resulting in duplicates in set $S_v$. After every exponentiation step, node $v$ has to locally remove these duplicates. As a result, when bounding the memory of a node, we have to take into account the momentary spike in global memory due to duplicates. This momentary spike can at most result in an $\hat{D}$ factor increase in the memory bounds.*

*Proof.* When node $v$ exponentiates, nodes $w$ send $S_{w \not\to r_w(v)}$ and not $S_w$. Consider node $x$ that $v$ has received via exponentiation, and consider the unique path $P_{vx}$ between $v$ and $x$. Since only nodes $w \in P_{vx}$ could have sent $x$ to $v$, and $|P_{vx}| \leq D \leq \hat{D}$, $x$ has at most $\hat{D}$ duplicates in $S_v$. ∎

Lemma 4.18. *In* CompressLightSubTrees*, the global memory never exceeds $O(n \cdot \hat{D}^3)$.*

*Proof.* The global memory $O(n \cdot \hat{D}^3)$ of ProbeDirections (Step 1(a)) follows from Lemma 4.12. Hence, we analyze the global memory excluding Step 1(a).

When all nodes are light, by Lemma 4.4, the size of the graph is $\leq 2n^{\delta/8}$. When taking duplicates into account (Observation 4.4), since $\hat{D} \leq n^{\delta/8}$, even if the whole graph is in the local memory of every node, this does not violate global memory constraints. For the rest of the proof assume that there is at least one heavy.

Consider an arbitrary iteration $j$ of the algorithm when there are heavy nodes. Define set $C_v \subseteq S_v$ as the set of nodes that $v$ has added to $S_v$ as a result of performing Exp in all iterations up to iteration $j$. Let $u$ be the parent of $v$ (according to the hypothetical rooting at $r$). For that $u$, define $C_{v\to u} \coloneqq C_v \cap G_{v\to u}$. For a node $v$, we have

$$|C_v| \leq |C_{v\to u}| + \sum_{w \in N(v)\setminus u} |S_{v\to w}| \leq |T(v,r)| \cdot \hat{D} + |T(v,r)| = (1+\hat{D})|T(v,r)|$$

The bound on $|C_{v\to u}|$ is obtained by applying Lemma 4.17 for the last iteration where $v$ has exponentiated in the direction of $u$, and the bound on the sum is by the definition of $T(v,r)$. Observe the crucial difference between $S_v$ and $C_v$. Set $S_v$ may contain nodes that are *not* a result of $v$ performing Exp, but rather the result of Step 1(d)ii of the algorithm, where some other node $w$ has added $v$ to $S_w$. However, this can result in at most a factor-2 overcounting for every node. Combining this with the duplicates of Observation 4.4 results in global memory

$$\hat{D} \cdot \sum_{v\in V} |S_v| = \hat{D} \cdot \sum_{v\in V} 2|C_v| \leq \hat{D} \cdot \sum_{v\in V} 2(1+\hat{D})|T(v,r)| = O(n \cdot \hat{D}^3),$$

where the bound on $\sum_{v\in V} |T(v,r)|$ is due to Lemma 4.5. ∎

Lemma 4.19. *In* CompressLightSubTrees*, the local memory of a node $v$ never exceeds $O(n^\delta)$.*

*Proof.* The local memory of ProbeDirections (Step 1(a)) follows from Lemma 4.12. Hence, we analyze the local memory excluding Step 1(a).

When all nodes are light, by Lemma 4.4, the size of the graph is $\leq 2n^{\delta/8}$. When taking duplicates into account (Observation 4.4), since $\hat{D} \leq n^{\delta/8}$, even if the whole graph is in the local memory of every node, this does not violate global memory constraints. For the rest of the proof assume that there is at least one heavy.

Consider the start of an arbitrary phase $i$. If $\deg(v) > n^\delta$, we defer the discussion to Lemma A.1 on the MPC implementation details. Assuming $\deg(v) \leq n^\delta$, we prove the claim by induction. During the algorithm, the size of the local memory is at most of order $|S_v| \cdot \hat{D}$ (the extra $\hat{D}$ factor is due to Observation 4.4). The claim clearly holds in the first iteration when $S_v$ is initialized as $N(v)$. Observe that if $\deg(v) > n^{\delta/8} + 1$, node $v$ becomes sad. Hence, we can further assume that $\deg(v) \leq n^{\delta/8} + 1$. Assume the claim holds in iteration $j$. We perform a case distinction on the different changes of $S_v$, and show that for a node $v$, it holds that $|S_v| = O(n^{7\delta/8})$, implying that $|S_v| \cdot \hat{D} = O(n^\delta)$ since $\hat{D} \leq n^{\delta/8}$.

- Step 1(c) and $|\mathsf{fullDirs}| = 1$,
  $|S_v|$ becomes at most $|S_{v\to\mathsf{fullDirs}}| + (\deg(v)-1)\cdot n^{\delta/8}\cdot\hat{D} \le 2n^{\delta/4} + (n^{\delta/8})^3 < n^{7\delta/8}$. The term $|S_{v\to\mathsf{fullDirs}}|$ has a (loose) upper bound of $2n^{\delta/2}$, since $v$ is not full. Observe that exponentiating in all directions except fullDirs yields $\le n^{\delta/8}\cdot\hat{D}$ nodes per direction by Lemma 4.12 $(i)$, and that $\hat{D} \le n^{\delta/8}$ by assumption.

- Step 1(d) and $|\mathsf{fullDirs}| = 0$,
  $|S_v|$ becomes at most $\deg(v)\cdot n^{\delta/8}\cdot\hat{D} \le (n^{\delta/8}+1)\cdot(n^{\delta/8})^2 < n^{7\delta/8}$. Observe that exponentiating in any direction yields $\le n^{\delta/8}\cdot\hat{D}$ nodes per direction by Lemma 4.12 $(ii)$ (fullDirs is empty), and that $\hat{D} \le n^{\delta/8}$ by assumption.

- Step 1(e),
  If a node $v$ becomes happy against $u$, $|S_v|$ becomes $|T(v,r)| + |S_{v\to u}| \le n^{\delta/8} + 2n^{\delta/4} < n^{7\delta/8}$, where $n^{\delta/8}$ is an upper bound for $|T(v,r)|$ since it is light, and $2n^{\delta/4}$ is (loose) upper bound on $|S_{v\to u}|$ since $v$ is not full.

Hence, the claim holds in iteration $j+1$. ◻

*Proof.* [Proof of Lemma 4.7 (Memory bounds)] The local memory bounds follow from Lemma 4.19, and the global memory bounds follow from Lemma 4.18. ◻

**4.3.3 Probing** Our probing procedure is an integral part of CompressLightSubTrees, as it steers the exponentiation of every node such that, informally, a node never learns a (significantly) larger neighborhood in the direction of the root (which is imagined only for the analysis), than in the direction of its subtree.

LEMMA 4.12. (ProbeDirections) *Consider an arbitrary iteration of algorithm* CompressLightSubTrees. *Then algorithm* ProbeDirections($\hat{D}$) *returns:*

*(i)* $\mathsf{fullDirs} \subseteq N(v)$ *such that if we were to exponentiate in all directions, we would obtain* $|S_{v\to u'}| > n^{\delta/8}$ *for all* $u' \in \mathsf{fullDirs}$ *and* $|S_{v\to u'}| \le n^{\delta/8}\cdot\hat{D}$ *for all* $u' \in N(v)\setminus\mathsf{fullDirs}$.

*(ii)* $\mathsf{largestDir} \in N(v)$ *(returned if* $\mathsf{fullDirs} = \emptyset$*) such that if we were to exponentiate in all directions, we would obtain* $|S_{v\to\mathsf{largestDir}}| \ge |S_{v\to u'}|$ *for all* $u' \in N(v)$ *and* $|S_{v\to\mathsf{largestDir}}| \le n^{\delta/8}\cdot\hat{D}$.

ProbeDirections *can be implemented in* $O(1)$ *low-space MPC rounds, using* $O(n\cdot\hat{D}^3)$ *global memory. It does not alter the state of* $S_v$ *for any node* $v$ *in the execution of* CompressLightSubTrees.

ProbeDirections($\hat{D}$)

1. For every neighbor $u \in N(v)$, compute $B_{v\to u} = \sum_{w\in S_{v\to u}} |S_{w\not\to r_w(v)}|$.
2. Define $\mathsf{fullDirs} \coloneqq \{u \in N(v) \mid B_{v\to u} \ge n^{\delta/8}\cdot\hat{D}\}$.
3. If $|\mathsf{fullDirs}| > 0$, define $\mathsf{largestDir} \coloneqq \emptyset$. Otherwise, let $u_{\max} = \arg\max_{u\in N(v)}\{B_{v\to u}\}$ and if $B_{v\to u_{\max}} \ge \hat{D}\cdot B_{v\to u'}$ for all $u' \in N(v)\setminus u_{\max}$
   (a) define $\mathsf{largestDir} \coloneqq u_{\max}$,
   (b) otherwise, perform $\mathsf{Exp}(N(v))$ and define $\mathsf{largestDir} \coloneqq \arg\max_{u\in N(v)}\{|S_{v\to u}|\}$.
4. Return fullDirs, largestDir.

LEMMA 4.20. *If a node* $v$ *were to perform* $\mathsf{Exp}(u)$ *for a neighbor* $u \in N(v)$*, it would hold that* $B_{v\to u}/\hat{D} \le |S_{v\to u}| \le B_{v\to u}$.

*Proof.* Recall the definition of $B_{v\to u}$. Let us compute how many times a node $w$ in $B_{v\to u}$ can be overcounted. Consider the unique path $P_{vw}$ from $v$ to a node $w$. Observe that out of the nodes in $S_{v\to u}$, node $w$ is in $S_{x\not\to v}$ only for nodes $x \in P_{vw}$. Since $|P_{vw}| \leq D+1 \leq \hat{D}+1$, any node $w$ is overcounted at most $\hat{D}$ times, completing the proof. ∎

*Proof.* [Proof of Lemma 4.12] Combining the condition $|B_{v\to u}| \geq n^{\delta/8}\cdot\hat{D}$ of Step 2 and the $\hat{D}$-factor overcounting of Lemma 4.20 proves the properties of fullDirs. The properties of largestDir hold by definition: in the case of Step 3(a), $u_{\max}$ is the largest direction by Lemma 4.20, and in the case of Step 3(b), we exponentiate and find the absolute values. Local memory is respected in Step 1, since node $v$ only aggregates an integer from every other node in $S_v$. More importantly, it is respected in Step 3: a node performing ProbeDirections has $\deg(v) < n^{\delta/8}+1$ (otherwise it is sad), Exp$(N(v))$ is only performed if all directions yield $\leq n^{\delta/8}\cdot\hat{D}$ nodes (fullDirs is empty), and we are promised that $\hat{D} \leq n^{\delta/8}$.

Global memory is respected by a clever observation similar to Lemma 4.18. Similarly to Section 4.3.2, assume we have a rooting at some node $r$, and that node $u$ is the parent of $v$. We want to bound the size of the resulting set $S_{v\to u}$ if node $v$ performs Exp$(N(v))$. In particular, we want to show that $|S_{v\to u}| \leq |S_{v\to w}|\cdot\hat{D}^2 \leq |T(v,r)|\cdot\hat{D}^2$ for *some* node $w \in N(v)\setminus u$. Towards contradiction, assume that $|S_{v\to u}| > |S_{v\to w}|\cdot\hat{D}^2$ for all $w \in N(v)\setminus u$. It must then hold that

$$B_{v\to u} \geq |S_{v\to u}| > |S_{v\to w}|\cdot\hat{D}^2 \geq B_{v\to w}/\hat{D}\cdot\hat{D}^2 = B_{v\to w}\cdot\hat{D}$$

for all $w \in N(v)\setminus u$ by Lemma 4.20. However, this implies that $u$ would have been chosen as $u_{\max}$, largestDir would have been defined as $u_{\max}$, and Exp$(N(v))$ would have never been performed; we have arrived at a contradiction. It holds that $|S_{v\to u}| \leq |T(v,r)|\cdot\hat{D}^2$, which bounds set $S_v$ of every node by $(\hat{D}^2+1)\cdot|T(v,r)|$, and by Lemma 4.5, the global memory is bounded by

$$\sum_{v\in V}|S_v| \leq (\hat{D}^2+1)\sum_{v\in V}|T(v,r)| \leq (\hat{D}^2+1)(D+1)\cdot n = O(n\cdot\hat{D}^3).$$

Regarding MPC implementation, ProbeDirections only performs Exp$(N(v))$ (implementability proven in Lemma A.1) and computes $B_{v\to u}$, which is only a modified version of Exp$(N(v))$: instead of nodes $w$ sending $S_{w\not\to r_w(v)}$ to node $v$, they only send $|S_{w\not\to r_w(v)}|$. ∎

**4.4 MAX-ID: Single Phase (CompressPaths)** Let us prove the following lemma, which allows us to compress all paths in the tree into single edges. This operation does not create new paths or disconnect the graph.

LEMMA 4.8. (CompressPaths) *For any tree $G$ and $\hat{D} \in [diam(G), n^{\delta/8}]$, CompressPaths$(G,\hat{D})$ returns the graph that is obtained from $G$ by replacing all paths of $G$ with a single edge. The algorithm runs in $O(\log\hat{D})$ low-space MPC rounds using $O(n\cdot\hat{D}^2)$ words of global memory.*

We describe a algorithm, which we denote as CompressPaths, and which we run on every path $P \subseteq G$. A path only includes consecutive degree-2 nodes. Similarly to CompressLightSubTrees, every node $v \in P$ has some set $S_v$ in its memory, which we initialize to $N_P(v)$. Every node performs Exp$(N(v))$ until $S_v$ no longer grows, whereupon, for every node $v$, it holds that $S_v = P$. The highest ID node $w \in P$ figures out the endpoints $x, y$ of path $P$ in $G$ (which either have degree 1 or $\geq 3$). Then, w.l.o.g., assume that $\mathrm{ID}(x) > \mathrm{ID}(y)$, whereupon $w$ compresses $P$ into $x$. By the definition of compression, node $w$ also creates edge $\{x, y\}$.

*Proof.* [Proof of Lemma 4.8] After performing CompressPaths, every node $v \in P$ learns path $P$, i.e., it holds that $S_v = P$, after $O(\log\hat{D})$ rounds, since the path is of length at most $\mathrm{diam}(G)$ and $\hat{D} \in [\mathrm{diam}(G), n^{\delta/8}]$. Node $w$ can learn $x, y$ by asking for the neighbors (that are in $G\setminus P$) of the leaf.

Because $|P| \leq D \leq \hat{D} \leq n^{\delta/8}$, the local memory of a node is bounded by $n^{\delta/4}$ (when taking Observation 4.4 into account). The global memory is respected since in the worst case, all nodes have at most $\hat{D}^2$ nodes in memory (when taking Observation 4.4 into account). Compressing and creating a new edge $\{x, y\}$ comprises of sending a constant sized message to both $x$ and $y$. Even in the case when $x$ or $y$ are endpoints to multiple paths, their total incoming message sizes are $O(\deg(x))$ and $O(\deg(y))$. The small caveat to this scheme is that if $\deg(x)$ or $\deg(y)$ are $> n^{\delta/8}$, we have to employ the aggregation tree structure as discussed in Lemma A.1. The implementation details of performing Exp can also be found in Lemma A.1. ∎

**4.5 MAX-ID: Single Reversal Phase** A single reversal phase consist of steps DecompressPaths and DecompressLightSubTrees. In the former, we essentially reverse CompressPaths, and in the latter, we reverse CompressLightSubTrees. We prove the following.

LEMMA 4.9. (DecompressPaths,DecompressLightSubTrees) *All nodes that were compressed by* CompressLightSubTrees *and* CompressPaths *can be decompressed by* DecompressLightSubTrees *and* DecompressPaths*, respectfully. The algorithms run in $O(1)$ low-space MPC rounds using $O(n)$ words of global memory.*

Let us introduce both steps formally.

- DecompressPaths. For every path $P$ that was compressed in phase $i$ into node $x$, node $x$ decompresses $P$ from itself.
- DecompressLightSubTrees. Every node $v$ that had compressed $T_v$ into a neighbor $u$ (or itself), decompresses $T_v$ from $u$ (or itself).

*Proof.* [Proof of Lemma 4.9] As long as the nodes $X$ that a node $v$ wants to decompress are in its local memory, both steps are clearly correct and implementable in $O(1)$ low-space MPC steps. Observe that all nodes $v$ that decompress a node set $X$, have at some point compressed set $X$ and hence, have had $X$ in local memory (in the form of $S_v$). By simply retaining set $X$ in memory until it is time to decompress, we fulfill the requirement. □

## 5 Connected Components (CC)

By Lemma 4.1, we can solve MAX-ID on any tree in $O(\log \hat{D})$ time using MAX-ID-Solver. The algorithm requires $O(m \cdot \hat{D}^3)$ words of global memory and value $\hat{D} \in [\text{diam}(G), n^{\delta/8}]$ as input. This section is mostly devoted to showing how to use MAX-ID-Solver to solve the connected components (CC) problem.

DEFINITION 5.1. (THE CC PROBLEM) *Given a graph with unique identifiers for each node, and disconnected components $C_1, \ldots, C_k$, every node $v \in C_i$ outputs the maximum identifier of $C_i$.*

Observe that MAX-ID-Solver actually solves CC for the case when the input graph is a single tree. We show how to extend MAX-ID-Solver to solve CC for forests, effectively proving the upper bounds of the following theorem.

THEOREM 5.1. (CONNECTED COMPONENTS) *Consider the family of forests. There is a deterministic low-space MPC algorithm to detect the connected components on this family of graphs. In particular, each node learns the maximum ID of its component. The algorithms runs in $O(\log D)$ rounds, where $D$ is the maximum diameter of any component. The algorithm requires $O(n+m)$ words of global memory, it is component-stable, and it does not need to know $D$. Under the 1 vs. 2 cycles conjecture, the runtime is asymptotically optimal.*

The proof is contained in Section 5.1 with references to subroutines from Sections 5.2 to 5.4. In Section 5.5, we show how to modify the algorithm of Section 5.1 to obtain a rooting.

**5.1 Proof of Theorem 1.1** There are three steps to extending MAX-ID-Solver and proving Theorem 1.1: (1) reducing the global memory to $O(n+m)$; (2) removing the need to know $\text{diam}(G)$ in order to give $\hat{D}$ as input; (3) generalizing it from trees to forests while maintaining component-stability. We address all steps separately.

1. By applying Lemma 5.1 before executing MAX-ID-Solver, we reduce the number of nodes in $G$ by a polynomial factor in $\hat{D}$. This reduces the global memory to a strict $O(n+m)$.
2. By employing the guessing scheme of Section 5.3, we perform multiple (sequential) executions of MAX-ID-Solver. Every execution is given a doubly exponentially growing guess for $\hat{D}$. The guessing scheme does not violate global memory $O(n+m)$ and results in a total runtime of $O(\log \text{diam}(G))$.
3. By the discussion in Section 5.4, we can execute MAX-ID-Solver on forests such that the runtime becomes $O(\log D)$, where $D$ is the largest diameter of any component. Moreover, when executing MAX-ID-Solver on forests, it is component-stable.

**5.2 CC: Pre- and Postprocessing** The aim of our *preprocessing* is to reduce the number of nodes in the input graph $G$ by a factor of $\text{poly}(\hat{D})$ (in fact $\hat{D}^3$ would suffice), resulting in graph $G'$. By executing MAX-ID-Solver on $G'$, we achieve a strict $O(n+m)$ global memory for one execution. When reducing the number of nodes, we must not disconnect the graph, and also keep the knowledge of the maximum ID inside the remaining graph.

After the connected components problem is solved on $G'$, we must extend the solution to the nodes in $G \setminus G'$ such that the solution is consistent. Extending the solution simply means informing every node in $G \setminus G'$ of $\underline{\text{ID}}$, which is the maximum identifier of the graph. We call this stage *postprocessing*. This section is devoted to proving the following lemma.

LEMMA 5.1. *Consider a tree $G$ with $n$ nodes. The number of nodes can be reduced by a factor of $\text{poly}(\hat{D})$ such that the resulting graph $G'$ remains connected, and one of the remaining nodes knows the maximum ID set $V_G \setminus V_{G'}$. If connected components is solved in $G'$, the solution can be extended to $G$. Both obtaining graph $G'$ from $G$ and extending the solution from $G'$ to $G$ takes $O(\log \hat{D})$ low-space MPC rounds using $O(n+m)$ words of global memory.*

Let us restate a known result that is going to be an essential tool in our preprocessing. We present a proof sketch to explicitly reason the memory bound.

LEMMA 5.2. ([23]) *There is an $O(1)$-round sublinear local memory (component unstable) MPC algorithm that, given a subset $U \subseteq V$ of nodes of a graph $G = (V, E)$ with $d_G(u) = 2$ for all $u \in U$, computes a subset $S \subseteq U$ that is an independent set in $G$ and satisfies $|S| \geq |U|/8$. The global memory used by the algorithm is $O(|U|) + O(|M| \cdot \log n) = O(n)$.*

*Proof.* [Proof Sketch] Consider the following random process: Each node marks itself with probability 1/2. If a node is marked and no neighbor is marked, it joins the set $S$, otherwise it does not. The probability of a node to be marked and not having any of its neighbors marked is 1/8. Thus, the expected size of $S$ is $|U|/8$. Further, note that this analysis still holds if the randomness for the nodes is 3-independent. $|U|$ coins that are 3-independent can be created from a bitstring of length $O(3 \cdot \log |U|) = O(\log n)$ (see Definition 5.2 and Theorem 5.2).

In order to deterministically compute the set $S$ we use the method of conditional expectation to compute a *good* bit string. For that purpose break the bitstring into $O(1)$ chunks of length at most $\delta/100 \log n$. Then we deterministically choose the bits on these segments such that the expected size of $S$ is remains $|U|/8$ conditioned on all already determined segments of random bits. To fix one segment introduce the indicator random variable $S_v$ that equals 1 if and only if $v \in S$. Let $\phi$ be the event that fixes to bitstring to what is already there and for $\alpha \in [n^{\delta/100}]$ let $\phi_\alpha$ be the event that the to be fixed segment equals $\alpha$. Knowing the Ids of its neighbors and the already fixed part of the bitstring, each machine can for each $v \in U$ that it holds compute the values $S_{v,\alpha} = \mathbb{E}[S_v \mid \psi = \alpha \wedge \phi]$. Then, nodes fix the current segment to the $\alpha_0$ such that minimizes $\sum_{v \in U} S_{v,\alpha}$. By the method of conditional expectation we have $\mathbb{E}[|S| \mid \psi_{\alpha_0} \wedge \phi] \leq |U|/8$. At the end the whole bitstring is fixed and we have deterministically selected an independent set $S$ satisfying $|S| \leq |U|/8$.

For an MPC implementation, we need to be able to globally, i.e., among all machines, to agree on the good bit string obtained from the method of conditional expectations. For this purpose, consider an aggregation tree structure, where the machines are arranged into a (roughly) $n^{\delta/2}$-ary tree Definition A.1 with depth $O(1/\delta)$. In this tree, each machine can choose the (locally) good bit segments (conditioned on the previous segments) of $\delta/100 \log n$ bits. Notice that there are at most $n^{\delta/100}$ such bit segments. Now, we can convergecast the expected size of $S$, given a segment, to the root. Then, the root can decide on the *good* (prefix of a) bit string. In each round, each machine receives a $n^{\delta/2} \cdot n^{\delta/100} \ll n^\delta$ bits, which fits the local memory. Since we have $O(n^{1-\delta})$ machines, the total memory requirement to store the bits is $O(n^{\delta/2+\delta/100+(1-\delta)}) = O(n)$.

DEFINITION 5.2. ([37]) *For $N, M, k \in \mathbb{N}$ such that $k \leq N$, a family of functions $\mathcal{H} = \{h : [N] \to [M]\}$ is $k$-wise independent if for all distinct $x_1, \ldots, x_k \in [N]$, the random variables $h(x_1), \ldots, h(x_k)$ are independent and uniformly distributed in $[M]$ when $h$ is chosen uniformly at random from $\mathcal{H}$.*

THEOREM 5.2. ([37]) *For every $a, b, k$, there is a family of $k$-wise independent hash functions $\mathcal{H} = \{h : \{0,1\}^a \to \{0,1\}^b\}$ such that choosing a random function from $\mathcal{H}$ takes $k \cdot \max\{a, b\}$ random bits.*

∎

Next, we introduce elementary operations Rake, Contract, which we use during preprocessing, and Insert, Expand, which we use during postprocessing. Operation Insert can be thought of as the reversal of Rake, and operation Expand as the reversal of Contract. Recall the definition of compression and decompression from the beginning of the section.

DEFINITION 5.3. *For a degree-1 node $v$ define the following two operations.*

- Rake($v$)*: Node $v$ compresses into its unique neighbor $u$*

+ Insert($v$)*: Node $v$ that underwent* Rake *decompresses from $u$.*

*For a degree-2 node $v$ define the following two operations.*

- Contract($v$)*: Node $v$ with neighbors $u$ and $w$ compresses into its highest ID neighbor.*

+ Expand($v$)*: Node $v$ that underwent* Contract *decompresses from $u$ (w.l.o.g. $ID(u) > ID(w)$).*

As long as we ensure that Insert($v$) and Expand($v$) are executed on nodes which have undergone Rake($v$) and Contract($v$) operations, respectively, we obtain the following observation.

OBSERVATION 5.1. *The operations* Rake($v$)*,* Insert($v$)*,* Contract($v$) *and* Expand($v$) *can be implemented in $O(1)$ low-space MPC rounds using $O(n+m)$ global memory on all nodes $v \in Z \subseteq V$ in parallel, if $Z$ is an independent set containing only degree-1 and degree-2 nodes.*

Let us introduce preprocessing and postprocessing formally. Note that preprocessing is performed directly on the input graph $G$, resulting in smaller graph $G'$. Whereas postprocessing is performed on a solved version of $G'$, i.e., with all nodes $v$ having $\mathsf{id}_v = \underline{\mathsf{ID}}$, resulting in a solved version of the input graph $G$. Both of the following routines use some constant $c$ in their runtime in order to reduce the number of nodes by a factor of $\hat{D}^c$. Initialize $G_0$ as the input graph $G$.

- Preprocessing. For $j = 0, \ldots, c \log \hat{D}$ iterations:
  1. Let $H$ be the subgraph induced by all degree-2 nodes in $G_j$. Compute an independent set $Z \in H$ of size at least $|H|/8$ using Lemma 5.2.
  2. Contract($v$) for every $v \in Z$.
  3. Rake($v$) for every degree-1 node $v$. If two leaves are neighbors, perform Rake only on the higher ID one.

- Postprocessing. For $j = c \log \hat{D}, \ldots, 0$ iterations:
  1. Insert($v$) for every node $v$ that performed Rake($v$) in iteration $j$ of Preprocessing.
  2. Expand($v$) for every node $v \in Z$ in iteration $j$ of Preprocessing.

*Proof.* [Proof of Lemma 5.1] Performing Preprocessing takes $O(\log \hat{D})$ time since Steps 1–3 can be performed in constant time. Let $G$ ($G'$) be the graph before (after) performing Preprocessing. Graph $G'$ has a poly($\hat{D}$) fraction less nodes that graph $G$ because a constant fraction of nodes in a tree have degree $\leq 2$, and we compress all degree $\leq 2$ nodes in the graph in each iteration.

Performing Postprocessing simply reverses Preprocessing while extending the current solution, so it has the same runtime as Preprocessing. Both Preprocessing and Postprocessing can be implemented in low-space MPC using $O(n+m)$ global memory, since other that Lemma 5.2, nodes only exchange constant sized messages with their neighbors, and don't store anything non-constant in local memory. ∎

**5.3 CC: Removing Knowledge of the Diameter** Our MAX-ID-Solver algorithm requires value $\hat{D} \in [D, n^{\delta/8}]$ as input, where $D$ denotes the diameter of the input tree. We show that we don't actually need to know $D$ in order to give value $\hat{D}$ as input. We achieve this by sequentially executing MAX-ID-Solver with a doubly exponential guess for $\hat{D}$ as follows.

$$\hat{D} = \hat{D}_1, \hat{D}_2, \ldots, \hat{D}_{\log\log n^{\delta/8}} = 2^{2^1}, 2^{2^2}, \ldots, 2^{2^{\log\log n^{\delta/8}}} .$$

We proceed with the next guess only if the previous has failed to terminate after a runtime of $O(\log \hat{D}_i)$. Detecting a failure within the given runtime, and making sure that a wrong guess does not violate memory constraints is a delicate affair, and is discussed in a separate paragraph. If all of our guesses fail, it must be that $D > n^{\delta/8}$. In this case, we can run the deterministic $O(\log n)$ time connected components algorithm of Coy and Czumaj [22] with the requirement that all nodes output the maximum ID of the component (their algorithm is component-stable for the same reasons the algorithm in this paper is). Hence, we can safely assume that $D \leq n^{\delta/8}$. Assuming that failure detection can be performed within $O(\log \hat{D}_i)$ and the given memory constraints, we show that the algorithm terminates successfully for some guess $\hat{D}$, and that the runtime resulting from our guessing scheme is acceptable. Eventually for some guess $l$, it holds that $D \leq \hat{D}_l$ and $\hat{D}_{l'} < D$ for all $l' < l$. In particular, it holds that $D \leq \hat{D}_l \leq D^2$. Since $\hat{D}_l \in [D, \min(n^{\delta/8}, D^2)] \subseteq [D, n^{\delta/8}]$, MAX-ID-Solver will terminate successfully for $\hat{D}_l$. Our guessing scheme results in a runtime of at most

$$\sum_{i=1}^{l} O(\log \hat{D}_i) \leq \left(\sum_{i=0}^{\infty} \frac{1}{2^i}\right) O(\log D) + O(\log D^2) = O(\log D) .$$

**Detecting Failure.** Let us consider the case when our guess $\hat{D}$ for the diameter is $< D$. It is possible for the algorithm to terminate even with a wrong diameter guess. However, we want to show that when the guess is wrong, we are able to detect it in $O(\log \hat{D})$ time even when the algorithm has not terminated. We also want to show that using a wrong diameter guess does not violate our memory constraints.

Our aim is to show that there exists a constant $c$ such that if, after $c \log \hat{D}$ rounds, the algorithm is unsuccessful (failed), we can manually terminate the execution and move on to the next diameter guess. The algorithm can be seen as unsuccessful, if after a constant number of phases, the graph is larger than a singleton (Lemma 4.11). Note that failure cannot be detected during preprocessing, since there we simply free an appropriate amount of memory for the actual algorithm. The exact number of phases can be deduced from Lemma 4.11. Phases consist of algorithms CompressLightSubTrees and CompressPaths which both have a runtime of $O(\log \hat{D})$ by Lemma 4.7 and Lemma 4.8, respectively. The exact constant in Lemma 4.7 can be computed from Lemmas 4.12, 4.15 and 4.16. If the diameter guess is wrong, it will simply result in removing too small subtrees, and possibly a graph larger than a singleton being left after the phases. The exact constant in Lemma 4.8 can be computed from its proof in Section 4.4. If the diameter guess is wrong, it may still result in CompressPaths terminating successfully. However, it may also result in nodes not learning the whole path they are in, which we can detect and manually terminate the execution.

It is left to show that for a wrong diameter guess, the local memory $O(n^\delta)$ and global memory $O(n+m)$ are not violated. Both cases are surprisingly straightforward. The former holds since all of the local memory arguments of the section are independent of $\hat{D}$ (they only use its upper bound $n^{\delta/8}$). The latter holds by performing Preprocessing for $3 \log \hat{D}$ iterations before every execution, since all of our lemmas use at most $O((n+m) \cdot \hat{D}^3)$ global memory.

**5.4 CC: Forests and Stability** When executing algorithm MAX-ID-Solver and the scheme developed in Section 5.1 on forests instead of trees, we have to consider how the disjoint components can affect each other with regards to runtime, memory, and component-stability (Definition 5.4).

Definition 5.4. (Component-stability, [24]) *A randomized MPC algorithm $A_{MPC}$ is component-stable if its output at any node $v$ is entirely, deterministically, dependent on the topology and IDs (but independent of names) of $v$'s connected component (which we will denote $CC(v)$), $v$ itself, the exact number of nodes $n$ and maximum degree $\Delta$ in the* entire *input graph, and the input random seed $\mathcal{S}$. That is, the output of $A_{MPC}$ at $v$ can be expressed*

*as a deterministic function $A_{\mathsf{MPC}}(CC(v), v, n, \Delta, \mathcal{S})$. A deterministic MPC algorithm $A_{\mathsf{MPC}}$ is component-stable under the same definition, but omitting dependency on the random seed $\mathcal{S}$.*

If we were to execute MAX-ID-Solver on a forest, nodes from disjoint components would never communicate with each other, rendering the runtime and memory arguments local. Hence, the algorithm is compatible with forests. In the scheme developed in Section 5.1 nodes from disjoint components communicate with each other only during preprocessing, when we employ the $O(1)$ time independent set algorithm of Lemma 5.2. Since the independent set is used to reduce the number of nodes *globally*, all of the runtime and memory arguments are still compatible with forests, as long as the given value $\hat{D}$ is in $[D, n^{\delta/8}]$, where $D$ is the largest diameter of any component.

What is left to argue is that if the input graph is a forest, MAX-ID-Solver and the scheme developed in Section 5.1 are component-stable. This however follows directly from the stability definition (Definition 5.4) and our problem definition (Definition 5.1), because we require nodes to output the maximum identifier of their component, which is fully independent of other components.

**5.5 Computing a Rooted Forest** In this section, we show how to use the connected components algorithm of Theorem 1.1, with minor adjustments, to root a forest. We prove the following.

THEOREM 5.3. (ROOTING) *Consider the family of forests with component-wise maximum diameter $D$. There is a deterministic low-space MPC algorithm that roots the forest in $O(\log D)$ rounds using $O(n+m)$ words of global memory, and it is component-stable.*

First, we execute the algorithm of Theorem 1.1 in order for every node to learn the maximum ID of its component. Using this knowledge, we execute a modified version of the same algorithm that roots (in a component-stable way) every component towards the single node (per component) that is left after $\ell$ phases. The modifications are the following.

1. Redefine compression and decompression as follows
   - Compress $X$ into $v$: remove $X$ (and its incident edges) from the graph. For any edge $\{x, y\}$ with $x \in X$ and $v \neq y \notin X$ we introduce a new edge $\{v, y\}$.
   - Decompress $X$ from $v$: Decompress $X$ from $v$: add $X$ (and its incident edges) back to the graph. If set $X$ is a subtree, orient the revived edges towards the single node to which the subtree is attached to. If set $X$ is a path, orient the revived edges in the same direction as edge $\{v, y\}$ that was added during the compression step of $X$ into $v$.
2. In the derandomization of Lemma 5.2, the process is run independently on each connected component. For each component, we use an aggregation tree (recall Definition A.1) that consist of nodes only in the corresponding component with the maximum ID node as a root. Then, the good bitstrings can be determined through the independent aggregation trees.

*Proof.* [Proof of Theorem 1.2] The first modification to the algorithm ensures a rooted forest. We prove it by induction, with the base case being a rooted graph $G_\ell = \{v\}$. If graph $G_i$ is rooted in the beginning of a reversal phase $i$, DecompressPaths extends the rooting of (some) single edges to paths, and DecompressLightSubTrees extends the rooting of (all) leaf nodes to subtrees, resulting in a rooted graph $G_{i-1}$ (recall that the reversal phase indices are in decreasing order). The same exact logic also holds for Postprocessing, where we extend the rooting to the nodes that were removed during Preprocessing.

The second modification ensures component-stability. Since the maximum ID node of each component is responsible only for its own component, we can choose the bitstring independently of the other components and create the broadcast tree (like in Lemma 5.2) for each component separately and independently. Then, the orientation of the rooting (i.e., which node will become the root) only depends on the topology and the maximum ID of the component, making it independent of the other components. ∎

## 6 Solving LCL Problems

In this section, we show a useful application of our forest rooting algorithm. In particular, we show that all problems contained in a wide class of problems that has been heavily studied in the distributed setting, called Locally Checkable Labelings (LCLs), can be solved in $O(\log D)$ deterministic rounds.

Informally, LCLs are a restriction of a class of problems called *locally checkable problems*. These problems satisfy that, given a solution, it is possible to check whether the solution is correct by checking the constant radius neighborhood around each node separately. Examples of these problems are classical problems such as maximal independent set, maximal matching, and $(\Delta+1)$-vertex coloring, but also more artificial problems, such that the problem of orienting the edges of a graph such that every node must have an odd number of outgoing edges.

The restriction that is imposed on locally checkable problems to obtain the class of LCLs is to require that the number of possible input and output labels that are required to define the problem must be constant, and moreover only graphs of bounded degree are considered. In the distributed setting, and in particular in the LOCAL model of distributed computing, LCLs have been extensively studied, see, e.g., [3, 9, 21, 12, 20, 10, 18, 16]. In particular, the imposed restriction makes it possible to prove very interesting properties on them, and to develop generic techniques to solve them. For example, we know that, if we restrict to forests, there are LCLs that can be solved in $O(1)$ rounds, there are LCLs that require $\Omega(\log^* n)$ rounds, but we also know that there is nothing in between, even if randomness is allowed (e.g., there are no LCLs with complexity $\Theta(\sqrt{\log^* n})$). Interestingly, techniques that have been developed to study LCLs have then often been extended and used to understand locally checkable problems in general (that is, problems that are not necessarily LCLs).

Since the LOCAL model is very powerful, and allows to send arbitrarily large messages, any solvable problem can be solved in $O(D)$ rounds. In this section, we provide an MPC algorithm for solving any solvable LCL problem on forests. The algorithm is deterministic, component-stable, and runs in $O(\log D)$ time in the low-space MPC model using $O(n+m)$ words of global memory (formal statement in Theorem 1.3). Moreover, our algorithm can be used *even for unsolvable LCLs*, that is, problems for which there exists some instance in which they are unsolvable. Hence, given any LCL, our algorithm produces a correct output on any instance that admits a solution, and it outputs "not solvable" on those instances where the LCL is not solvable.

THEOREM 1.3. (LCLS ON TREES) *All LCL problems on forests with maximum component diameter $D$ can be solved in $O(\log D)$ time in the low-space MPC model using $O(n+m)$ words of global memory. The algorithm is deterministic and does not require prior knowledge of $D$.*

Notice that it is enough to prove Theorem 1.3 for rooted forests, since we can first root the forest by spending the same runtime and memory (Theorem 1.1), and then solve the LCL.

The remaining of the section is structured as follows: we start by giving a formal definition of LCLs (Section 6.1); we proceed by providing a high-level overview of our algorithm (Section 6.2); then, we provide the definition of the concept of "compatibility tree", that will be useful later (Section 6.3); in Section 6.4 we give the explicit algorithm, called LCLSolver; in Sections 6.6 to 6.10 we show some properties of the subroutines used in LCLSolver, and we bound the time complexity of each of them; finally, we put things together and prove the main theorem of this section in Section 6.11.

We note that, for the sake of simplicity, algorithm LCLSolver is described for trees, but we will show in Section 6.11 that it can also be executed on forests.

**6.1 Locally Checkable Labelings** Locally Checkable Labeling (LCL) problems have been introduced in a seminal work of Naor and Stockmeyer [35]. The definition they provide restricts attention to problems where the goal is to label nodes (such as vertex coloring problems), but they remark that a similar definition can be given for problems where the goal is to label edges (such as edge coloring problems). A modern way to define LCL problems that captures both of the above types of problems (and combinations thereof) consists of labeling of *half-edges*, i.e., pairs $(v,e)$ where $e$ is an edge incident to vertex $v$. Let us first formally define half-edge labelings, and then provide this modern LCL problem definition.

DEFINITION 6.1. (HALF-EDGE LABELING) *A* half-edge *in a graph $G=(V,E)$ is a pair $(v,e)$, where $v \in V$, and $e=\{u,v\} \in E$. We say that a half-edge $(v,e)$ is incident to some vertex $w$ if $v=w$. We denote the set of half-edges of $G$ by $H=H(G)$. A* half-edge labeling *of $G$ with labels from a set $\Sigma$ is a function $g\colon H(G) \to \Sigma$.*

We distinguish between two kinds of half-edge labelings: *input labelings*, that are labels that are part of the

input, and *output labelings*, that are provided by an algorithm executed on input-labeled instances. Throughout the paper, we will assume that any considered input graph $G$ comes with an input labeling $g_{\text{in}} \colon H(G) \to \Sigma_{\text{in}}$ and will refer to $\Sigma_{\text{in}}$ as the *set of input labels*; if the considered LCL problem does not have input labels, we can simply assume that $\Sigma_{\text{in}} = \{\perp\}$ and that each half-edge is labeled with $\perp$.

Informally, LCLs are defined on bounded-degree graphs, where each node may have in input a label from a constant-size set $\Sigma_{\text{in}}$ of labels, and must produce in output a label from a constant-size set $\Sigma_{\text{out}}$ of labels. Then, an LCL is defined through a set of locally checkable constraints that must be satisfied by all nodes.

Definition 6.2. (LCL) *An LCL problem $\Pi = (\Sigma_{\text{in}}, \Sigma_{\text{out}}, C, r)$ is defined as follows:*

- *$\Sigma_{\text{in}}$ and $\Sigma_{\text{out}}$ are sets of constant size that represent, respectively, the possible input and output labels.*
- *The parameter $r$ is a constant called* checkability radius of $\Pi$.
- *$C$ is a set of constant size, containing allowed neighborhoods. Each element $c_i = (G_i, v_i)$ of $C$, where $G_i = (V_i, E_i)$, is such that:*
  - *$G_i$ is a graph satisfying that $v_i \in V_i$ and that the eccentricity of $v_i$ in $G_i$ is at most $r$;*
  - *Every half-edge of $G_i$ is labeled with a label in $\Sigma_{\text{in}}$ and a label in $\Sigma_{\text{out}}$.*

Definition 6.3. (Solving an LCL) *In order to solve an LCL on a given graph $G = (V, E)$ where to each element $(v, e) \in V \times E$ is assigned an input label from $\Sigma_{\text{in}}$, we must assign to each element $(v, e) \in V \times E$ an output label from $\Sigma_{\text{out}}$ such that, for every $v \in V$, it holds that $(G_r(v), v) \in C$, where $G_r(v)$ is the subgraph of $G$ induced by nodes at distance at most $r$ from $v$ and edges incident to at least one node at distance at most $r - 1$ from $v$.*

Example 6.1. (Maximal Independent Set) *In the maximal independent set problem, the goal is to select an independent set of nodes that cannot be extended. That is, selected nodes must not be neighbors, and non-selected nodes must have at least one neighbor in the set.*

*For this problem, $\Sigma_{\text{in}} = \{\perp\}$. Then, we can use two possible output labels, 1 to indicate nodes that are in the set, and 0 to indicate nodes that are not in the set. Hence, $\Sigma_{\text{out}} = \{0, 1\}$. Finally, we need to define $r$ and $C$. For this problem, it is sufficient to pick $r = 1$. In $C$, we put all possible pairs $(G, v)$ satisfying the following:*

- *$G$ is a star centered at $v$;*
- *$G$ has at most $\Delta$ leaves (and there can be 0 leaves);*
- *For each node in $G$, either all incident half-edges are output labeled 0, or all incident half-edges are output labeled 1;*
- *If $v$ is labeled 1, then all leaves are labeled 0;*
- *If $v$ is labeled 0, then at least one leaf is labeled 1.*

*Hence, the idea is that MIS can be checked by just inspecting the radius-1 neighborhood of each node, which is a star, and we just list all stars that are valid.*

While Definition 6.2 gives an easy way to define problems, such a definition is not the most convenient for proving statements about LCLs. In order to make our proofs more accessible, we consider an alternative definition of LCLs, called node-edge formalism. It is known that, on trees and forests, any LCL defined as in Definition 6.2 can be converted, in a mechanical way, into an LCL described by using this formalism, such that the obtained LCL has the same asymptotic complexity of the original one [10].

Definition 6.4. (Node-edge-checkable LCL) *In this formalism, a problem $\Pi$ is a tuple $(\Sigma_{\text{in}}, \Sigma_{\text{out}}, C_V, C_E)$ satisfying the following:*

- *As before, $\Sigma_{\text{in}}$ and $\Sigma_{\text{out}}$ are sets of constant size that represent, respectively, the possible input and output labels;*

- *$C_V$ and $C_E$ are both sets of multisets of pairs of labels, where each pair is in $\Sigma_{\text{in}} \times \Sigma_{\text{out}}$, and multisets in $C_E$ have size 2.*

Definition 6.5. (Solving a node-edge checkable LCL) *Solving an LCL given in this formalism means that we are given a graph $G = (V, E)$ where to each element $(v, e) \in V \times E$ is assigned a label $i_{v,e}$ from $\Sigma_{\text{in}}$, and to each element $(v, e) \in V \times E$ we must assign a label $o_{v,e}$ from $\Sigma_{\text{out}}$ such that:*

- *For every node $v \in V$ it holds that the multiset $M_v = \{(i_{v,e}, o_{v,e}) \mid e \text{ is incident to } v\}$ satisfies $M_v \in C_V$;*
- *For every edge $e \in E$ it holds that the multiset $M_e = \{(i_{v,e}, o_{v,e}) \mid e \text{ is incident to } v\}$ satisfies $M_e \in C_E$.*

Hence, in the node-edge checkable formalism, we are given a graph where each half-edge (that is, an element from $V \times E$) is labeled with a label from $\Sigma_{\text{in}}$, the task is to label each half-edge from a label from $\Sigma_{\text{out}}$, and the LCL constraints are expressed by listing tuples of size at most $\Delta$ representing allowed configurations for the nodes, and tuples of size 2 representing allowed configurations for the edges. In [10] it has been shown that any LCL $\Pi$ defined on trees or forests can be converted into a node-edge checkable LCL $\Pi'$ satisfying that the complexity of $\Pi$ and $\Pi'$ differ only by an additive constant. Hence, for the purposes of this work, we can safely restrict our attention to node-edge-checkable LCLs.

Example 6.2. (Maximal Independent Set) *Sometimes, defining an LCL in the node-edge checkable formalism is non-trivial. MIS is an example of problems in which the conversion requires a bit of work (it can be done mechanically, though, as shown in [10]). We hence use MIS as an example for this formalism.*

*As before, $\Sigma_{\text{in}} = \{\perp\}$, and hence when listing the elements in $C_V$ and $C_E$ we will not specify the input labels. This time, it is not actually possible to use just 2 labels as output. In fact, we define $\Sigma_{\text{out}} = \{0, 1, \text{P}\}$. Then, $C_V$ contains all multisets of size at most $\Delta$ satisfying that:*

- *All elements are 1, or*
- *one element is P and all the others are 0.*

*Then, $C_E = \{\{1, 0\}, \{1, \text{P}\}, \{0, 0\}\}$. In other words, nodes in the MIS output 1 on all their incident half-edges, nodes not in the MIS output P on one incident half-edge and 0 on all the others. The label P is used to prove maximality. That is, nodes not in the set must point to one neighbor in the set by using the label P. In fact, on the edge constraint, P is only compatible with 1. Observe that, given a solution for the standard MIS problem, a solution for this variant can be produced with just one round of communication.*

**6.2 Overview** On a high level, our algorithm works as follows. We describe it from the point of view of a single node, and for a single tree of the forest. Firstly, we root the tree, obtaining that each node knows the edge connecting it to its parent. Then, the algorithm proceeds in phases, and in total the number of phases is going to be a constant that depends on the amount of memory available to the machines. In each phase, we *compress* the tree into a smaller tree, as follows:

- all subtrees containing less than a fixed amount of nodes are compressed to their root;
- all paths are compressed into a single edge.

Each phase is going to require $O(\log D)$ time. In other words, this part of our algorithm works similar to the standard rake-and-compress algorithm. Moreover, while compressing the tree, we maintain some information about the LCL that we are trying to solve. This information is called *compatibility tree*.

The compatibility tree, for each node and for each edge, keeps track of the possible configurations that they can use. At the beginning, for each node, these configurations correspond to the configurations in $C_V$ that are compatible with the given input, and for each edge, these configurations correspond to the configurations in $C_E$ that are compatible with the given input.

When compressing a subtree into a single node, we update the list of the configurations usable on that node, in such a way that each configuration satisfies the following: if the node uses it, then it is possible to assign a labeling on the subtree compressed into that node, in such a way that, for each node and edge in the compressed subtree we use only configurations allowed by the compatibility tree before the compression.

Similarly, when compressing a path, for the new edge that we add, we store a list of configurations satisfying that, if we label the first and last half-edge of the removed path with the labels of the configuration, then we can complete the compressed path by only using configurations allowed by the compatibility tree before the compression.

At the end, we obtain that the whole tree is recursively compressed on a single node $v$. If the compatibility tree does not allow any configuration for $v$, then we know that the LCL is unsolvable. Otherwise, we can pick an arbitrary configuration allowed by the compatibility tree and assign it to $v$. By performing this operation, we know that we can safely put back the paths and subtrees that were compressed on $v$ and have the guarantee that we can label them using only allowed configurations. Hence, we again proceed in phases, where we put back compressed paths and subtrees in the opposite order in which they have been compressed, and each time we assign labels allowed by the compatibility tree. At the end, we obtain that the whole tree is labeled correctly, and hence the LCL is solved.

**6.3 Compatibility Tree** A compatibility tree is an assignment of sets of allowed configurations to nodes and edges, where this time configurations are not just multisets, but they are tuples. In other words, we may allow a node to use a configuration, but only if the labels of that configuration are used in a very specific order.

Definition 6.6. (Compatibility Tree) *A compatibility tree of a tree $G = (V, E)$ is a pair of functions $\phi$ and $\psi$, where $\phi$ maps each node $v \in V$ into a set of tuples of size at most $\Delta$, and $\psi$ maps each edge $e \in E$ into a set of tuples of size 2.*

In order to specify how the compatibility tree is initialized, it is useful to first assign an order to the edges incident to each node, and to the nodes incident to each edge. This ordering is called port numbering assignment. Observe that an arbitrary port numbering assignment can be trivially computed in 1 round of communication.

Definition 6.7. (Port Numbering) *A node port numbering is a labeling of every half-edge satisfying that, for each node $v$, half-edges incident to $v$ have pairwise distinct values in $\{1, \ldots, \deg(v)\}$. An edge port numbering is a labeling of every half-edge satisfying that, for each edge $e$, half-edges incident to $e$ have pairwise distinct values in $\{1, 2\}$. A port numbering is the union of a node port numbering and an edge port numbering.*

Assume that the tree $G$ is already provided with a port numbering. The compatibility tree of $G$ is initialized as follows. For each node $v$, $\phi(v) = \{(\ell_1, \ldots, \ell_{\deg(v)}) \mid \{(i_1, \ell_1), \ldots, (i_{\deg(v)}, \ell_{\deg(v)})\} \in C_V\}$, where $i_j$ is the input assigned to the half-edge incident to $v$ with node port number $j$. For each edge $e$, $\psi(e) = \{(\ell_1, \ell_2) \mid \{(i_1, \ell_1), (i_2, \ell_2)\} \in C_E\}$, where $i_j$ is the input assigned to the half-edge incident to $e$ with edge port number $j$. In other words, we initialize $\phi$ and $\psi$ with everything that is allowed by the constraints of the problem, in all possible orders that are compatible with the given input.

We can observe that, by construction, $\phi$ and $\psi$ still encode the original problem. In other words, we can now forget about $C_V$ and $C_E$, and try to find a labeling assignment that is valid according to $\phi$ and $\psi$. We make this observation more formal in the following statement.

Observation 6.1. *The LCL problem $\Pi$ is solvable if and only if there is a labeling $g_{\text{out}} : H \to \Sigma_{\text{out}}$ that solves $\Pi$ that satisfies that:*

- *For each node $v$, let $\ell_j$ be the label assigned by $g_{\text{out}}$ to the half-edge incident to $v$ with port number $j$. It must hold that $(\ell_1, \ldots, \ell_{\deg(v)}) \in \phi(v)$.*
- *For each edge $e$, let $\ell_j$ be the label assigned by $g_{\text{out}}$ to the half-edge incident to $e$ with port number $j$. It must hold that $(\ell_1, \ell_2) \in \psi(e)$.*

On a high level, when compressing a subtree into a node $v$, we will redefine $\phi(v)$ and discard some tuples. The discarded tuples are the ones satisfying that, if node $v$ uses such a configuration, there is no way to complete the labeling of the subtree in a valid way. Similarly, when compressing a path, we will define $\psi(e)$, where $e$ is the new (virtual) edge that we use to replace the path, in such a way that, if $\psi(e)$ contains the tuple $(\ell_1, \ell_2)$ and we label the first half-edge of the compressed path with $\ell_1$ and the last half-edge with $\ell_2$, then we can correctly complete the labeling inside the path. Here it should become clear why we use tuples and not just multisets: it may be that $v$ can use a label on the half-edge connecting it to one child (because that subtree can be competed

by starting with that label), but the same label cannot be used on the half-edge connecting $v$ to a different child. A similar situation could happen on a compressed path: it could be that it is possible to label $\ell_1$ the half-edge connecting the first endpoint to the path and $\ell_2$ the half-edge of the second endpoint to the path, but not vice versa.

In the algorithm, we will solve the problem $\Pi$ in the tree obtained by compressing some subtrees into single nodes, and some paths into single edges. We now formally define what it means to partially solve an LCL $\Pi$ w.r.t. a compatibility tree $(\phi, \psi)$. Observe that, in a tree $G$ obtained after performing some compression steps, a node $v$ may have a degree that is smaller than the size of the tuples given by $\phi(v)$, that always have size equal to the original degree of $v$, denoted by $\mathrm{origdeg}(v)$, and hence the ports incident to $v$ may be just a subset of $\{1, \ldots, \mathrm{origdeg}(v)\}$.

Definition 6.8. (Partially solving an LCL w.r.t. the compatibility tree) *Let $G$ be a tree, and let $(\phi, \psi)$ be a compatibility tree for $G$. A solution for $\Pi$ that is correct according to $\phi$ and $\psi$ is a labeling $g_{\mathrm{out}}$ satisfying that:*

- *For each node $v$, let $\mathrm{origdeg}(v)$ be the size of the tuples given by $\phi(v)$, and let $P(v) \subseteq \{1, \ldots, \mathrm{origdeg}(v)\}$ be the subset of ports of $v$ that are present in $G$. For each $j \in P(v)$, let $\ell_j$ be the label assigned by $g_{\mathrm{out}}$ to the half-edge incident to $v$ with port number $j$. There must exist labels $\ell_k$, for all $k \in \{1, \ldots, \mathrm{origdeg}(v)\} \setminus P(v)$, such that it holds that $(\ell_1, \ldots, \ell_{\mathrm{origdeg}(v)}) \in \phi(v)$.*
- *For each edge $e$, let $\ell_j$ be the label assigned by $g_{\mathrm{out}}$ to the half-edge incident to $e$ with port number $j$. It must hold that $(\ell_1, \ell_2) \in \psi(e)$.*

In other words, solving the LCL in the tree obtained by performing some compression steps, means to pick, for each node, a configuration allowed by $\phi$, in such a way that all edges that are still present have a configuration allowed by $\psi$.

**6.4 The Algorithm** Let $\Pi = (\Sigma_{\mathrm{in}}, \Sigma_{\mathrm{out}}, C_V, C_E)$ be the considered LCL problem, and let $G_0 = G$ denote a rooted input tree with root $r$. The high-level idea of our approach is to first initialize $\phi_0$ and $\psi_0$ as the functions $\phi$ and $\psi$ shown in Section 6.3. Then, we perform the following distinct parts.

**Steps 1–2 of LCLSolver.** From $G_0$, we iteratively derive a sequence $G_1, G_2, \ldots, G_t$ of smaller trees until eventually, for some $t = O(1)$, it holds that $G_t$ consists of a single node (the root $r$). In the meanwhile, we also update the compatibility tree, and compute $\phi_j$ and $\psi_j$ for all $0 < j \leq t$. The sequence is derived such that $G_j$ $(0 < j \leq t)$ is obtained from $G_{j-1}$ by first compressing all subtrees of size $\leq n^{\delta/2}$ into their respective roots (we refer to the roots of the subtrees and not (necessarily) the actual root node $r$), and then compressing all paths into single edges. Throughout the sequence of compatibility trees, we maintain the following property, which we prove in Lemmas 6.3 and 6.6.

Claim 6.1. *Let $1 \leq j \leq t$. If there exists a correct solution for $G_{j-1}$ according to $\phi_{j-1}$ and $\psi_{j-1}$ (w.r.t. Definition 6.8), then there exists also a correct solution for $G_j$ according to $\phi_j$ and $\psi_j$ (w.r.t. Definition 6.8). Moreover, given any correct solution for $G_j$, we can transform it into a correct solution for $G_{j-1}$.*

**Steps 3–4 of LCLSolver.** For all $0 \leq i \leq t$, we define $\dot{G}_i$ to be $G_i$ where to each node $v$ is assigned a configuration $c(v) \in \phi_i(v)$ in such a way that the assignment $c$ induces a labeling $g_{\mathrm{out}}$ that is correct according to $\phi_i$ and $\psi_i$ (w.r.t. Definition 6.8). The sequence is derived such that $\dot{G}_i$ is obtained from $\dot{G}_{i+1}$ by decompressing the subtrees and paths that were compressed when $G_{i+1}$ was obtained from $G_i$, and simultaneously solving the problem, which is possible by Claim 6.1. Finally, the solution on $\dot{G}_0$ is a solution for $\Pi$ on $G$ by Observation 6.1.

**The Algorithm.** Let us now formally present the algorithm LCLSolver (along with its subroutines) that solves any LCL on rooted trees in $O(\log D)$ time. Note that the subroutines are state-changing functions, i.e., they modify their input graphs.

LCLSolver$(\Pi, G(V, E))$

1. Initialize $\phi_0$ and $\psi_0$ according to Section 6.3.

2. Initialize phase counter $j \leftarrow 0$. Repeat the following until the graph is a singleton.
   (a) CountSubtreeSizes($G$)
   (b) GatherSubtrees($G$)
   (c) CompressSubtrees($G$)
   // compress all subtrees of size $\leq n^{\delta/2}$ into single nodes, also compute $\phi_{j+1}$
   (d) AdvancedCompressPaths($G$)
   // compress all paths into single edges, also compute $\psi_{j+1}$
   (e) Update $j \leftarrow j + 1$
3. Set $c(v)$ to be an arbitrary element of $\phi_j(v)$, where $v$ is the obtained singleton.
   // the graph $\dot{G}_j = \dot{G}$ is obtained
4. Initialize repetition counter $k \leftarrow j$. Repeat the following while $k \geq 0$.
   (a) DecompressPaths($\dot{G}$)
   // decompress the paths from phase $k$
   (b) DecompressSubtrees($\dot{G}$)
   // decompress the subtrees from phase $k$, the graph $\dot{G}_{k-1}$ is obtained
   (c) Update $k \leftarrow k - 1$

Observe that the phase counter is incremented, while the repetition counter is decremented, which is inline with the indexing used in the previous high-level overview. Next, we give a brief introduction to the subroutines, before defining them formally in the following subsections. Recall that $T(v)$ denotes the subtree that is rooted at a node $v$ such that $v$ belongs to $T(v)$ and $G \setminus T(v)$ is connected.

- CountSubtreeSizes: Every node $v$ learns either the exact size of $T(v)$ or that $|T(v)| > n^{\delta/2}$. In the former case, $v$ marks itself as *light*, and in the latter case, as *heavy*. If a heavy node has a light child, it remarks itself as a *local root*.
- GatherSubtrees: Every local root $v$ learns $T(u)$ for every light child $u$.
- CompressSubtrees: Every local root $v$ checks, for each half-edge $h$ connecting it to a light child $u$, what are the possible labelings of $h$ that allow to complete the labeling of $T(u)$ in such a way that it is valid according to $\phi$ and $\psi$. Then, $v$ compresses all of these trees into itself, and updates $\phi$ in such a way that any LCL solution on the remaining graph can be extended to a solution on $T(u)$ for every light child $u$. This is done according to the computed possible labelings of the half-edges.
- AdvancedCompressPaths: For every path $P$ with some endpoints $u$ and $w$ (both have either degree 1 or $\geq 3$), compress $P$ into a new edge $\{u, w\}$, and assign an arbitrary edge port numbering to this edge. The value of $\psi(\{u, w\})$ is then defined in a way that any LCL solution on the edge $\{u, w\}$ can be extended to a solution on $P$. After performing CompressPaths, there are no degree-2 nodes left, which will be crucial for the analysis.
- DecompressPaths: The LCL problem on the input graph is solved. In repetition $k$, we decompress all paths that were compressed during phase $k$. While decompressing, we extend the solution to the paths.
- DecompressSubtrees: The LCL problem on the input graph is solved. In repetition $k$, every local root of phase $k$ decompresses all subtrees that it compressed during phase $k$. While decompressing, we extend the solution to the subtrees.

The following sections are rather self-contained and correspond to a specific subroutine that is called by LCLSolver (in the order they are called). They are written from a node's point of view, with the proofs intertwining correctness, runtime, and MPC details.

**6.5 Solving LCLs:** CountSubtreeSizes During the execution of the subroutine, every node $v$ maintains the following variables:

- $i$: iteration counter
- $s(v)$: size of $T(v)$ until depth $2^i$ ($v$ is at depth 0)
- $C(v)$: set of all descendant nodes (if any) of $v$ at depth $2^i$ ($v$ is at depth 0).

The aim of the subroutine is to detect all heavy nodes, i.e., nodes which have a subtree of size $> n^{\delta/2}$ rooted at them. This can be thought of as a preprocessing step for GatherSubtrees. All nodes are initially marked as active.

CountSubtreeSizes($G$)

Each node $v$ initializes: $C(v) \leftarrow$ set of children of $v$ in $G$, $s(v) \leftarrow |C(v)|$, and $i \leftarrow 0$.

1. Repeat the following steps until $C(v) = \emptyset$ for every node $v$.
   (a) If $v$ is active and all $u \in C(v)$ are also active, $v$ updates
   - $C(v) \leftarrow \bigcup_{u \in C(v)} C(u)$
   - $s(v) \leftarrow s(v) + \sum_{u \in C(v)} s(u)$.

   Otherwise, $v$ marks itself as heavy, and it becomes inactive.
   (b) If $s(v) > n^{\delta/2}$, $v$ marks itself as heavy, and it becomes inactive. Heavy nodes update $C(v) \leftarrow \emptyset$.
   (c) All active nodes update $i \leftarrow i + 1$.

Non-heavy nodes marks themselves as light.

Upon termination, for every heavy node $v$ it holds that $|T(v)| > n^{\delta/2}$. Note that the ancestors of heavy nodes are also heavy; heavy nodes induce a single connected component in $G$.

LEMMA 6.1. (CountSubtreeSizes) *Every node $v$ learns either the exact size of $T(v)$ or that $|T(v)| > n^{\delta/2}$. The algorithm terminates in $O(\log D)$ low-space MPC deterministic rounds using $O(n+m)$ words of global memory.*

*Proof.* We first show that every active node $v$ maintains the correct values for $s(v)$ and $C(v)$ throughout the algorithm. In iteration $i = 0$, values $s(v)$ and $C(v)$ are correct by initialization. During iteration $i$ in Step 1(a), $v$ updates its values only if $v$ together with all of its descendants in $C(v)$ are active, resulting in the correct values for iteration $i + 1$ by construction (see Figure 2). A node $v$ marks itself as heavy only when $s(v) > n^{\delta/2}$ or when one of its descendants is heavy. Both conditions imply that $|T(v)| > n^{\delta/2}$. If neither conditions are met and $C(v) = \emptyset$ at some point, then the value $s(v)$ for node $v$ is the exact size of $T(v)$ and $v$ marks itself light.

The algorithm terminates in $O(\log D)$ iterations (with each iteration taking $O(1)$ MPC rounds), since an active node knows $T(v)$ until depth $2^i$ in iteration $i$, and the depth of a tree is $D$. Observe that for every light node $v$, it holds that $|C(v)| \leq s(v) \leq n^{\delta/2}$. Hence, local memory is never violated, because when a node updates $C(v)$ in Step 1(a), the resulting set is of size at most $n^{\delta/2} \cdot n^{\delta/2}$ nodes. Also, storing value $s(v)$ takes only $O(\log n)$ bits. Global memory is never violated, since, by design, a node $u$ is only kept in the set $C(\cdot)$ of exactly one node.

Algorithm CountSubtreeSizes can be thought of as a modified version of graph exponentiation where nodes only keep track of the furthest away descendants. For the communication in Step 1(a) to be feasible, the set $C(v)$ is simply a set of IDs corresponding to the desired nodes. Observe that the communication in Step 1(a) is always initialized by $v$, and not by the descendants in $C(v)$ (nodes in $C(v)$ don't even know the ID of $v$). This is feasible, because, by design, every node has at most one ancestor that initializes communication. ∎

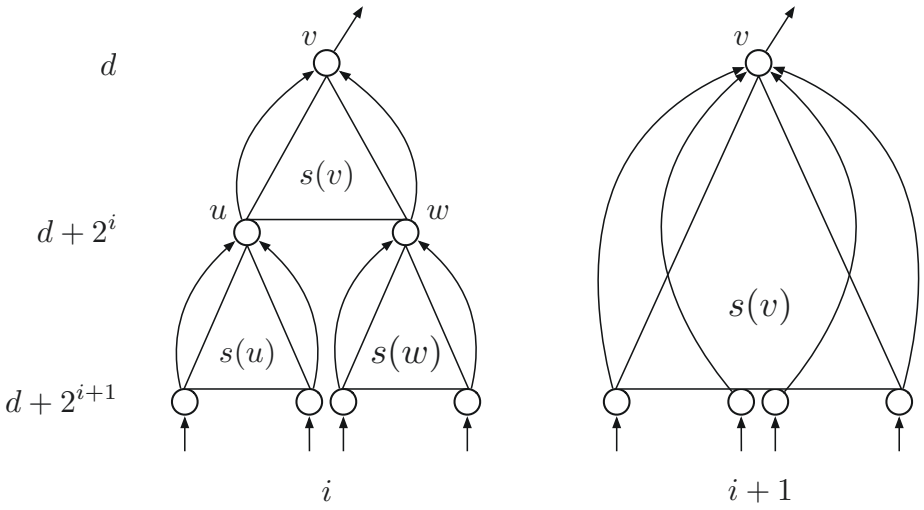


Figure 2: An illustration of an update in Step 1(a) of CountSubtreeSizes. The intuition is that node $v$ learns the size of the partial subtrees hanging from every child in $C(v)$. The values $d$ and $d + 2^i$ on the left refer to the depth of nodes $v$ and $u, w$, respectively, with regards to the whole tree. The oriented edges are not the actual edges of $G$, but rather a representation of sets $C(\cdot)$. The incoming edges of node $v$ are incident to the nodes in set $C(v)$ in both iterations $i$ and $i + 1$. The value $s(v)$ in iteration $i + 1$ is simply the sum of $s(v)$, $s(u)$ and $s(w)$ from iteration $i$.

**6.6 Solving LCLs:** GatherSubtrees After executing CountSubtreeSizes, by Lemma 6.1, every node knows if it is heavy or light. Moreover, every heavy node $v$ knows if it has a light child or not. If so, node $v$ remarks itself from heavy to *local root*. If there are no local roots in $G$, mark the actual root of the tree as a local root. During algorithm GatherSubtrees, heavy nodes do nothing, and all other nodes (including local roots) maintain the following variables:

- $i$: iteration counter
- $C(v)$: a subset of descendant nodes.

The procedure is as follows (see Figure 3 for an example).

GatherSubtrees($G$)

Each node $v$ initializes: $C(v) \leftarrow$ set of light children of $v$ in $G$, and $i \leftarrow 0$.

1. Repeat the following steps until $C(v)$ for every local root $v$ consist of the union of subtrees $T(u)$ for every light child $u$.
   (a) Every local root $v$ updates $C(v) \leftarrow C(v) \cup \bigcup_{u \in C(v)} C(u)$.
   (b) Every light node $w$ updates $C(w) \leftarrow \bigcup_{u \in C(w)} C(u)$.
   (c) Update $i \leftarrow i + 1$.

We phrase the algorithm in terms of the subtrees of the light children of a local root $v$, instead of the subtree of $v$ directly, and we do this for a simple reason: a local root $v$ may have children that are also local roots, in which case, $v$ does not want to learn anything in their direction.

LEMMA 6.2. (GatherSubtrees) *Every local root has gathered the subtree $T(u)$ for every light child $u$. The algorithm terminates in $O(\log D)$ low-space MPC deterministic rounds using $O(n + m)$ words of global memory.*

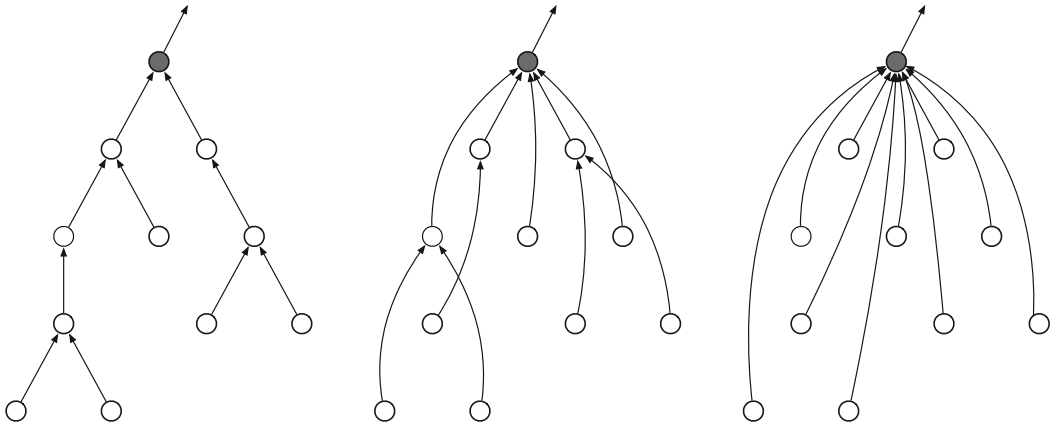


Figure 3: Three iterations of GatherSubtrees, with a local root marked black. The directed edges are not necessarily the actual edges of $G$, but rather a representation of sets $C(\cdot)$. The incoming edges of a node $v$ are incident to the nodes in set $C(v)$. The figure illustrates how local roots behave differently than other nodes: local roots aggregate all descendants, while other nodes replace current ones with new ones.

*Proof.* By design, the set $C(v)$ of a local root $v$ contains $T(u)$ until depth $2^i - 1$ for every light child $u$ in iteration $i$. Since the depth of a tree is $D$, after at most $O(\log D)$ iterations, for every local root $v$, $C(v)$ contains $T(u)$ for every light child $u$.

Observe that $|T(u)| \leq n^{\delta/2}$ for every light node $u$. This implies that $|C(v)| \leq n^{\delta}$ for every local root $v$, since it has a constant number of children (the maximum degree of the graph is constant). Hence, local memory is respected. Global memory is never violated, since by design, a node $u$ is only kept in set $C(\cdot)$ of exactly one node.

Similarly to CountSubtreeSizes, algorithm GatherSubtrees can be thought of as a modified version of graph exponentiation. However, as opposed to CountSubtreeSizes, algorithm GatherSubtrees actually gathers the whole subtrees into the memory of preselected nodes (local roots). Similarly to CountSubtreeSizes, we store IDs in the sets $C(\cdot)$ in order for the communication in Step 1 to be feasible. Also, the communication is made possible due to every node having at most one ancestor that initializes the communication. □

**6.7 Solving LCLs:** CompressSubtrees After executing GatherSubtrees, by Lemma 6.2, every local root has gathered the IDs of the nodes in the subtree $T(u)$ for every light child $u$.

CompressSubtrees($G$)

1. Perform the following step for every light child $u$ of every local root $v$. Denote $e^* = (u, v)$. Every local root $v$ gathers the topology of $T(u)$, along with $\phi(w)$ for every node $w$ in $T(u)$ and $\psi(e)$ for every edge $e$ in $T(u)$. Every local root $v$ computes the set of labels $L(e^*)$ satisfying that, by labeling the half-edge $(v, e^*)$ with a label in $L(e^*)$, it is possible to complete the labeling in $T(u)$ by only using configurations allowed by $\phi$ and $\psi$.
2. Every local root $v$ updates $\phi(v)$ by possibly discarding some tuples. Let $P(v)$ be the set of ports of $v$ connecting it to light nodes, and let $e_i$ be the edge reached from $v$ by following port $i$. A tuple $(\ell_1, \ldots, \ell_d)$ is kept in $\phi(v)$ if and only if $\ell_i \in L(e_i)$ for all $i \in P(v)$.

LEMMA 6.3. *Let $\phi$ and $\psi$ be the compatibility tree before performing* CompressSubtrees*, and let $\phi'$ be the updated compatibility tree after performing* CompressSubtrees*. Let $G'$ be the resulting graph after performing*

CompressSubtrees. *The following holds.*

- *If there exists a correct solution for the input graph $G$ according to $\phi$ and $\psi$, then there exists also a correct solution for $G'$ according to $\phi'$ and $\psi$.*
- *Given any correct solution for $G'$ according to $\phi'$ and $\psi$, it can be transformed into a correct solution for $G$ according to $\phi$ and $\psi$.*

*Proof.* In order to show the first statement, suppose that there is a correct solution for $G$. Notice that, since $G'$ is a subgraph of $G$, if, for every $v \in G'$, we use the tuple in $\phi(v)$ of the correct solution in $G$, then we get a correct solution for $G'$. Hence, we need to ensure that the tuple used by $v$ is in $\phi'(v)$. But this is exactly what we do: every tuple excluded from $\phi(v)$ in Step 2 is not part of any correct solution for $G$, and hence the first statement holds. For the second statement, observe that from the definition of $G'$, it follows that any correct solution for $G'$ provides a partial solution for $G$ (all labels are fixed except the ones in the compressed subtrees), and this partial solution is part of a correct solution for $G$. Hence, a correct solution for $G$ can be obtained by extending the provided solution to the compressed subtrees. An extension is guaranteed to exist, since all non-extendable tuples of $\phi(v)$ were removed previously in Step 2. Note that this extension can be performed by all local roots simultaneously, since there are no dependencies between subtrees. ∎

LEMMA 6.4. (CompressSubtrees) *The algorithm terminates in $O(1)$ time in the low-space MPC model and uses $O(n+m)$ words of global memory.*

*Proof.* Gathering the topology of $T(u)$, along with $\phi(w)$ for every node $w$ in $T(u)$ and $\psi(e)$ for every edge $e$ in $T(u)$ is possible: by Lemma 6.2, the local root $v$ knows the IDs of all nodes in $T(u)$, and hence node $v$ can simply gather all incident edges from all nodes in $T(u)$ in constant time, and reconstruct $T(u)$ locally. This does not break any memory constraints, since $v$ receives every edge from at most two nodes, the number of edges is bounded by the number of nodes, and the sets $\phi(w)$ and $\psi(e)$ are of constant size. Computing the sets $L(\cdot)$ does not require communication, and can be done locally in constant time. Finally, the removal (contraction) of nodes and updating the set $\phi(v)$ also takes constant time, concluding the runtime proof. Moreover, the global memory is not violated, since similarly to GatherSubtrees, a node $u$ is gathered by only one local root $v$. ∎

**6.8 Solving LCLs:** AdvancedCompressPaths The aim of this algorithm is to compress all paths into single edges while retaining the compatibility information of the paths, i.e., if the problem is solved, the solution can be extended to the paths that were compressed. As opposed to the previous subroutines, AdvancedCompressPaths does not capitalize on anything that is done by the previous routines.

We begin with a slight detour and first show how all degree-2 nodes can compute their distance to the highest ID endpoint with the following algorithm. To keep things simple, we present an algorithm for a single path $H$ with two endpoints of degree 1.

CountDistances($G$)

1. Define degree-2 nodes as *internal* nodes, degree-1 nodes as *endpoints*, and the higher ID endpoint as the *head*. The head is denoted by $h$. Assign weight $w(e) \leftarrow 1$ for every edge $e$. Repeat the following steps until all internal nodes share a weighted edge with both endpoints; the weight of the edge equals the distance.
   (a) Every internal node with incident edges $e = \{u, v\}$ and $e' = \{v, w\}$, removes* $e$ and $e'$ from $H$ and replaces them with a new edge $e'' = \{u, w\}$ and sets $w(e'') \leftarrow w(e) + w(e')$.
   *If $u$ (or $w$) is an endpoint, node $v$ does not remove $e$ (or $e'$).

LEMMA 6.5. (CountDistances) *Every degree-2 node knows its distance to both endpoints. The algorithm does not require prior knowledge of $D$, terminates in $O(\log D)$ low-space MPC rounds using $O(n+m)$ words of global memory.*

*Proof.* The weight of each edge $\{v, u\}$ in the graph equals the number of edges between $v$ and $u$ in the original graph. The base case being evident from the initialization of the weight of each edge, and the induction step from the update step $w(e'') \leftarrow w(e) + w(e')$. Since the shortest path from either endpoint to any other node in the path decreases by a factor of at least 3/2, CountDistances terminates in $O(\log D)$ time.

Creating edges and communicating through them can be done in constant time in MPC, since storing an edge equals storing the ID of the neighbor. Observe that every internal node keeps exactly two edges in memory. In order to not break their local memory, endpoints do not keep track of any edges. Since all nodes keep at most two edges in memory, global memory is respected. □

Next, we show that, by using the distances computed with CountDistances, we can compress paths of all lengths in $O(\log D)$ time, while respecting both local and global memory. Recall that for a path, $h$ denotes its highest ID endpoint.

AdvancedCompressPaths($G$)

1. Set $i \leftarrow 0$ and execute the following steps for every path $H$ until it consists of one edge.
   (a) Define an MIS set $Z_i \coloneqq \{v \in H \mid \deg(v) = 2 \text{ and } d_G(v, h) \text{ is } \textit{not} \text{ divisible by } 2^{i+1}\}$.
   (b) Every node $v \in Z_i$ with incident edges $e = \{u, v\}$ and $e' = \{v, w\}$ removes $e$ and $e'$ from $G'$ and replaces them with a new edge $e'' = \{u, w\}$ (with port 1 connected to $u$ and port 2 connected to $w$). Furthermore, for the new edge, $v$ sets $\psi(e'')$ to the set of all tuples $(\ell_1, \ell_2)$ satisfying that there exist two labels $x$, $y$ satisfying the following. Let $p_1$ be the port connecting $e$ to node $u$, let $p_2$ be the port connecting $e$ to $v$, let $p_3$ be the port connecting $v$ to $e$, let $p_4$ be the port connecting $v$ to $e'$, let $p_5$ be the port connecting $e'$ to $v$, and let $p_6$ be the port connecting $e'$ to $w$:
      - There is a tuple in $\psi(e)$ with label $\ell_1$ in position $p_1$ and label $x$ in position $p_2$;
      - There is a tuple in $\phi(v)$ with label $x$ in position $p_3$ and label $y$ in position $p_4$;
      - There is a tuple in $\psi(e')$ with label $y$ in position $p_5$ and label $\ell_2$ in position $p_6$.
   (c) Update $i \leftarrow i + 1$.

From the perspective of the nodes $u$ and $w$, the new edge $e''$ replaces the old edges $e$ and $e'$, respectively. In other words, if $u$ was connected through port $j$ to edge $e$, now it is connected through port $j$ to edge $e''$. Notice that, the reason for which, at each step, we compute an MIS, is that, if MIS nodes replace their two incident edges of the path with a single edge, we still obtain a (shorter) path as a result.

**Consecutive MIS.** Executing CountDistances gives us the means to compute consecutive maximal independent sets in AdvancedCompressPaths, which is not exactly obvious nor easily attainable using other means. If we were to compute an MIS directly with, e.g., Linial's [32] algorithm in every iteration of Step 2, we would end up with a total runtime of $O(\log D \cdot \log^* n)$. An alternative approach would be to employ the component-unstable $O(1)$-time algorithm that computes an independent set of size $\Omega(n/\Delta)$ by [24]. This approach also fails for multiple paths, since the algorithm in [24] does not give the guarantee that a constant fraction of nodes in *all* paths join the independent set, leading to a total runtime of $O(\log n)$. To summarize, CountDistances is a novel approach to a very non-trivial problem, yielding component stability and a sharp $O(\log D)$ runtime.

LEMMA 6.6. *Let $\phi$ and $\psi$ be the compatibility tree before performing* AdvancedCompressPaths*, and let $\psi'$ be the updated compatibility tree after performing* AdvancedCompressPaths*. Let $G'$ be the resulting graph after performing* AdvancedCompressPaths*. The following holds.*

- *If there exists a correct solution for the input graph $G$ according to $\phi$ and $\psi$, then there also exists a correct solution for $G'$ according to $\phi$ and $\psi'$.*
- *Given any correct solution for $G'$ according to $\phi$ and $\psi'$, it can be transformed into a correct solution for $G$ according to $\phi$ and $\psi$.*

*Proof.* Assuming that $Z_i$ is indeed an MIS, the first statement follows from the fact that acting nodes (i.e., MIS nodes) are never neighbors and every set $\psi(\{u, w\})$ that an acting node $v$ creates only discards configurations that do not correspond to valid solutions for the subpath $(u, v, w)$. The second statement holds by the definition of labels $\ell_1, \ell_2, x, y$, since we can perform the process in reverse.

Let us show that $Z_i$ constitutes an MIS. For any $i$, observe that the distances of the remaining nodes constitute all multiples of $2^i$ up until some number (the length of the path). Hence, every second node is not divisible by $2^{i+1}$ and joins $Z_i$, proving the statement. ∎

LEMMA 6.7. (AdvancedCompressPaths) *There are no degree-2 nodes left in the graph. The algorithm terminates in $O(\log D)$ low-space MPC rounds using $O(n + m)$ words of global memory.*

*Proof.* Since $Z_i$ constitutes an MIS, every path shortens by a constant factor. After $O(\log D)$ iterations, every path is compressed into a single edge. Every iteration consists of a constant number of communication rounds, every node uses a constant amount of memory, and compressing paths into edges never creates new degree-2 nodes. ∎

**6.9 Solving LCLs:** DecompressPaths Assuming that the problem of interest is solved in the current graph, we essentially reverse AdvancedCompressPaths and iteratively extend the solution from certain edges to the paths that were previously compressed into those edges. By "the problem is solved in the current graph" we simply mean that the output labels of the half-edges in the current graph satisfies Definition 6.8.

DecompressPaths($\dot{G}$)

1. All nodes that performed AdvancedCompressPaths in phase $k$, know the last iteration $i$ and can perform the following until $i = 0$.
   (a) Every node $v \in Z_i$ learns the fixed half-edge labels $(\ell_1, \ell_2)$ assigned to $e'' = (u, w)$ ($e''$ is the edge $v$ had created). Node $v$ removes $e''$ from the graph and replaces it with $e = (u, v)$ and $e' = (v, w)$ (edges $e$ and $e'$ are the edges $v$ had removed). Furthermore, $v$ assigns half-edge labels $\ell_1, x$ to edge $e$ and labels $y, \ell_2$ to edge $e'$ such that the labeling satisfies $\psi(e)$, $\phi(v)$, and $\psi(e')$.
   (b) Update $i \leftarrow i - 1$

LEMMA 6.8. (DecompressPaths) *The LCL problem on the graph is solved according to Definition 6.8, and the graph has the same node and edge sets as $G$ in phase $k$ before executing* AdvancedCompressPaths. *The algorithm terminates in $O(\log D)$ low-space MPC rounds using $O(n + m)$ words of global memory.*

*Proof.* All we do is reversing the steps of AdvancedCompressPaths and extending the solution for $\dot{G}$ to the decompressed paths, resulting in a correct solution on the graph that has the same node and edge sets that we had before the compression. Since the computation of the solution is done locally, and extending the solution requires a constant amount of memory and communication, the lemma follows from Lemmas 6.6 and 6.7. ∎

**6.10 Solving LCLs:** DecompressSubtrees The assumption for this algorithm, similarly to DecompressPaths, is that the LCL problem on the graph is solved correctly according to Definition 6.8. We also assume, and keep the invariant, that we do not only know the partial output assignment given to $\dot{G}$, but we also know, for each node $v$ of $G$, the tuple $c(v) \in \phi(v)$ assigned to it. This is especially useful at the beginning, when we have the root that is a singleton, and hence has no incident edges in the current graph, but we still want to know how to label its incident half-edges after decompressing the subtrees rooted at its children.

DecompressSubtrees($\dot{G}$)

1. Every local root $v$ of phase $k$ decompresses every subtree $T(u)$ compressed into it during phase $k$, while simultaneously solving the LCL problem on $T(u)$.

LEMMA 6.9. (DecompressSubtrees) *The LCL problem on the graph is solved according to Definition 6.8, and it has the same node and edge sets as $G$ in phase $k$ before executing* CompressSubtrees. *The algorithm terminates in $O(1)$ low-space MPC rounds using $O(n+m)$ words of global memory.*

*Proof.* The first statement follows from Lemma 6.3, since the solution for $\dot{G}$ can be extended to the decompressed trees, resulting in a correct solution on the graph that has the same node and edge sets that we had before the compression. For each node $u$ in the decompressed trees, we store in $c(u)$ the tuple used to label its incident half-edges. The runtime follows from Lemma 6.4, and the memory is respected trivially. ∎

**6.11 Proof of Theorem 1.3** By the lemmas in Sections 6.6 to 6.10, the problem is solved in the original forest, and all subroutines of LCLSolver have time complexity $O(\log D)$ in the low-space MPC model and use $O(n+m)$ words of global memory. All of the subroutines are clearly deterministic. What is left to prove is that

(i) after a constant number of phases in Step 2, the graph is reduced to a single node;

(ii) after a constant number of repetitions in Step 4, the graph is expanded to its original form;

(iii) if the input graph is a forest, the algorithm is component-stable, and the runtime becomes $O(\log D_{\max})$, where $D_{\max}$ denotes the maximum diameter of any component.

*Proof.* [Proof of (i)] The proof is very similar to the proof of Lemma 4.11, but we restate the claims for completeness. Let us recall what effectively happens during a phase. There are only two subroutines that alter the graph: in CompressSubtrees, all subtrees of size $\leq n^{\delta/2}$ are compressed into the first ancestor $v$ with a subtree of size $> n^{\delta/2}$; then, in AdvancedCompressPaths, all paths are compressed into single edges, leaving no degree-2 nodes in the graph. Let $G_j$ and $n_j = |G_j|$ denote the graph and the size of the graph at the beginning of phase $j$, respectively. We claim that after one phase, the number of nodes in the graph drops by a factor of $\Theta(n^{\delta/2})$. Observe, that after CompressSubtrees every leaf $w$ in the graph corresponds to a subtree of size $\geq n^{\delta/2}$ that was removed. Moreover, the same holds also after AdvancedCompressPaths. Hence,

$$\begin{aligned} n_j &\geq n_{j+1} + |\{w \in G_{j+1} \mid \deg_{G_{j+1}}(w) = 1\}| \cdot n^{\delta/2} \\ &> n_{j+1} + n^{\delta/2} \cdot n_{j+1}/2 \cdot \\ &= n_{j+1}(1 + n^{\delta/2}/2), \end{aligned}$$

implying that

$$n_{j+1} \leq \frac{n_j}{1 + n^{\delta/2}/2} \ .$$

The first strict inequality stems from the fact that there are no degree-2 nodes left after phase $j$, and hence the number of leaf nodes in $G_{j+1}$ is strictly larger that $n_{i+1}/2$. It is clear that after $O(1/\delta)$ phases, the graph is reduced to one node. ∎

*Proof.* [Proof of (ii)] Let us recall what effectively happens during a repetition. Both subroutines DecompressPaths and DecompressSubtrees alter the graph by decompressing the paths and subtrees that were compressed previously in some phase. Hence, the number of repetitions is equal to number of phases, which is constant. ∎

*Proof.* [Proof of (iii)] During the algorithm, the only communication between the components happens in order to start the subroutines in synchrony, which does not affect the LCL solution. It does however affect the runtime, since smaller components may be stalled behind larger components. Hence, in all runtime arguments, $D$ can be substituted with $D_{\max}$. ∎

**Extension to unsolvable LCL problems.** If the LCL problem is unsolvable, we can detect it in the following way. If, during any phase of Step 2, a local root $v$ ends up with an empty set of tuples in $\phi(v)$, the original LCL problem must be unsolvable. Node $v$ can then broadcast to all nodes in the graph to output label $\perp$ on their incident half-edges, indicating that there is no solution to the LCL problem.

## 7 Conditional Hardness Results

In this section, we show that our algorithm for solving all LCLs is optimal, assuming a widely believed conjecture about MPC. By earlier work, we consider the following more convenient problem that is also hard under the conjecture. We note that, due to technical reasons, our problem definition is slightly different to the one in [26]. Following in the footsteps of previous work, we will show that our version of the problem is also hard under the 1 vs. 2 cycles conjecture.

Definition 7.1. (The $D$-diameter $s$-$t$ path-connectivity problem) *Consider a graph that consists of a collection of paths of diameter $O(D)$, for some parameter $D$ satisfying $D \in \Omega(\log n)$ and $D = n^{o(1)}$. Given two special nodes $s$ and $t$ of degree 1 in the graph, the algorithm should provide the following guarantee: If $s$ and $t$ are in the same connected component, then the algorithm should output YES. If $s$ and $t$ are in different connected components, the algorithm should output NO.*

Lemma 7.1. *Assuming that the 1 vs. 2 cycles conjecture holds, there is no deterministic low-space MPC algorithm with $\mathrm{poly}(n)$ global memory to solve the $D$-diameter $s$-$t$ path-connectivity problem in $o(\log D)$ rounds.*

*Proof.* On a high level, we show that we can use an algorithm for the $D$-diameter $s$-$t$ path-connectivity problem to reduce the size of the given cycles by a multiplicative factor $D$, unless the given cycles are already too small. We then show that, by recursively applying this algorithm, we obtain a solution for the 1 vs. 2 cycles cycles problem.

In more detail, we are given a graph $G$ that is either one or two cycles, where each cycle is of length at least $n/2$. Let $D$ be in $\Omega(\log n)$ and in $n^{o(1)}$. We proceed in phases, starting in phase $i = 0$. We assume that at the beginning of phase $i$ the graph $G$ contains at least $(n/D^i)/2^i$ nodes and at most $(n/D^i)\cdot 2^i$ nodes, and we guarantee that at the end of phase $i$ the graph contains at least $(n/D^{i+1})/2^{i+1}$ nodes and at most $(n/D^{i+1})\cdot 2^{i+1}$ nodes. If the graph contains two cycles, this factor-$D$ reduction will actually independently hold for the size of each cycle. This is performed by running the algorithm $A$ that solves the $D$-diameter $s$-$t$ path-connectivity problem. We stop when the number of nodes is $n^{o(1)}$, which requires $O(\log_D n)$ phases. Then, we can spend $o(\log n)$ rounds to solve the problem with known techniques (e.g., [14]). If the $D$-diameter $s$-$t$ path-connectivity problem could be solved in $o(\log D)$ rounds, we would obtain a total running time of $O(\log_D n)\cdot o(\log D) + o(\log n) = o(\log n)$, which violates the 1 vs. 2 cycles conjecture. We now explain a single phase of the algorithm.

In each phase $i$, we maintain the invariant that, if there are two cycles, the larger one contains at most $4^i$ times the nodes of the smaller one. Assume that at the beginning of phase $i$ there are at least $(4^i+1)cD\log n$ nodes, for a sufficiently large constant $c$. If it is not the case, then we are done, because the number of nodes is in $n^{o(1)}$.

Sample the nodes in $G$ with probability $1/D$ and turn each sampled node *inactive*. At the end of the phase, only inactive nodes will remain, and by a standard Chernoff bound, with high probability, the number of inactive nodes is at least a factor $D/2$ and at most a factor $2D$ smaller than the original amount of nodes. Moreover, this holds independently on each cycle, and hence the ratio of the sizes of the obtained cycles can increase by at most a factor 4, hence maintaining the invariant.

We now show an upper bound on the length of the obtained paths, induced by active nodes. Consider a sequence of $c'D\log n$ nodes, for some sufficiently large constant $c'$. The probability that none of them is sampled is $(1-1/D)^{c'D\log n}$, and hence, with high probability, each path has length $O(D\log n)$. Moreover, since the shortest cycle has at least $cD\log n$ nodes, then, by fixing $c$ sufficiently larger than $c'$, we obtain that each cycle contains at least one sampled node with high probability, and hence $G'$ is a collection of paths, as required.

We create many instances of the $D$-diameter $s$-$t$ path-connectivity problem from these paths as follows. Fix a node $u$ with degree 1 in $G'$. We set $u \coloneqq s$ and create an instance of $D$-diameter $s$-$t$ path-connectivity for each possible choice of $t \neq s$, where $t$ is also a degree 1 node. Notice that there can be at most $n$ of such choices. Furthermore, we do the same construction for all possible choices of $s$, which results in $O(n^2)$ instances of the $D$-diameter $s$-$t$ path-connectivity problem.

Suppose now that we have a deterministic $o(\log D)$ time algorithm $A$ to solve the $D$-diameter $s$-$t$ path-connectivity problem. The paths have length $O(D\log n)$, and hence running this algorithm requires $o(\log(D\log n))$

rounds, which, by the assumption on $D$, is still in $o(\log D)$. Run $A$ independently on each of the $O(n^2)$ instances of the $D$-diameter $s$-$t$ path-connectivity problem. On an instance where $s$ and $t$ are on the same path, the algorithm returns YES and otherwise NO. Hence, we can derive which endpoints in $G'$ are on the same path in $o(\log D)$ time.

Then, we create a new instance of the 1 vs. 2 cycles cycle problem as follows. For each pair $s$ and $t$ on the same path, we create a virtual edge between the inactive neighbors of $s$ and $t$ and remove the active nodes. The number of nodes decreases at least by a factor $D/2$ and at most by a factor $2D$, as required. ∎

Since a connected component algorithm clearly solves the $D$-diameter $s$-$t$ path-connectivity problem, we obtain the following corollary.

COROLLARY 7.1. *Assuming the 1 vs. 2 cycles conjecture, there is no low-space memory MPC algorithm to solve connected components in $o(\log D)$ rounds on forests.*

We now show that we can define an LCL problem $\Pi$ for which we can convert any solution into a solution for the problem of Definition 7.1 in constant time. This implies a conditional lower bound of $\Omega(\log D)$ for $\Pi$, implying also that our generic solver, that runs in $O(\log D)$ rounds, is optimal. Instead of defining $\Pi$ by defining $C_V$ and $C_E$ formally, which makes it difficult to parse the definition, we provide a human understandable description of the constraints.

- The possible inputs of the nodes are 0 or 1. In the instances that we create, all nodes will be labeled 0, except for $s$, which will be labeled 1.
- The possible outputs are on edges, and every edge needs to be either oriented or unoriented.
- All nodes of degree 2 must have either both incident edges unoriented, or both incident edges oriented. If they are oriented, one must be incoming and the other outgoing.
- Any node of degree 1 with input 1 must have its incident edge oriented.
- Any node of degree 1 with input 0 must have its incident edge either unoriented, or oriented incoming.

We can observe some properties on the possible solutions for this problem:

- The edges of a path are either all oriented or all unoriented.
- The edges of a path containing only nodes with input 0 must all be unoriented, because a path needs to be oriented consistently, and endpoints with input 0 must have their edge oriented incoming.
- All the edges of a path containing an endpoint with input 1 must be oriented.

We can use an algorithm for $\Pi$ to solve the problem of Definition 7.1 as follows. By giving 0 as input to all nodes except $s$, and 1 to $s$, and solving $\Pi$, we obtain a solution in which only the other endpoint of the path containing $s$ has an oriented incident edge, and we can hence check if this node is $t$. Since $\Pi$ is an LCL, we obtain the following.

THEOREM 7.1. *Assuming the 1 vs. 2 cycles conjecture, there is no low-space memory MPC algorithm to solve any solvable LCL in $o(\log D)$ rounds on forests.*

## A MPC Implementation Details

Initially, before executing any algorithm, the input graph of $n$ nodes and $m$ edges is distributed among the machines arbitrarily. By applying Definition A.1, we can organize the input such that every node and it's edges are hosted on a single machine, or, in the case of high degree, on multiple consecutive machines.

DEFINITION A.1. (AGGREGATION TREE STRUCTURE, [13]) *Assume that an MPC algorithm receives a collection of sets $A_1, \ldots, A_k$ with elements from a totally ordered domain as input. In an aggregation tree structure for $A_1, \ldots, A_k$, the elements of $A_1, \ldots, A_k$ are stored in lexicographically sorted order (they are primarily sorted by the number $i \in \{1, \ldots, k\}$ and within each set $A_i$ they are sorted increasingly). For each $i \in \{1, \ldots, k\}$ such that the elements of $A_i$ appear on at least 2 different machines, there is a tree of constant depth containing the*

*machines that store elements of $A_i$ as leafs and where each inner node of the tree has at most $n^{\delta/2}$ children. The tree is structured such that it can be used as a search tree for the elements in $A_i$ (i.e., such that an in-order traversal of the tree visits the leaves in sorted order). Each inner node of these trees is handled by a separate additional machine. In addition, there is a constant-depth aggregation tree of degree at most $n^{\delta/2}$ connecting all the machines that store elements of $A_1, \ldots, A_k$.*

This section is dedicated to showing how MAX-ID-Solver can be implemented in the low-space MPC model. We only cover routine CompressLightSubTrees, since the implementation details for CompressPaths, DecompressLightSubTrees, and DecompressPaths are simple, and included in the corresponding proofs.

In the proof of CompressLightSubTrees, we have reasoned that the local memory of a node never exceeds $O(n^\delta)$, and that the total memory never exceeds $O(n \cdot \hat{D}^3)$. However, we have to also ensure that the low-space MPC's communication bandwidth of $O(n^\delta)$ is respected throughout the routines (Lemma A.1). Also, we have to address the possibility of a node having degree $> n^\delta$, since we work with arbitrary degree trees.

If, during some iteration of CompressLightSubTrees, the degree of a node is $> n^\delta$, it is clearly heavy, and does not partake in the ongoing iteration. In fact, if the degree is $> n^{\delta/8} + 1$, it is also heavy and does not partake. Hence, in the following lemma, we can assume that every node $v$ and its edges are hosted on a single machine, and that $\deg(v) \leq n^{\delta/8} + 1$.

LEMMA A.1. *The following routines can be performed in $O(1)$ low-space MPC rounds:*

1. *A node can detect whether it is happy or full,*
2. *$\mathsf{Exp}(X), X \subseteq N(v)$,*
3. *If node $v$ is added in $S_w$ for some $w$, $v$ is able to add $w$ to $S_v$.*

*Proof.* We prove the three statements separately. All three statements have the a common technical difficulty: it is possible for node $v$ to be included in $S_w$ for some $w$, such that $w \notin S_v$, which causes communication bandwidth congestion. We address this common issue shortly, after reasoning about the separate challenges of each routine.

1. Since the property of being full depends solely on the size of $S_v$, it can be computed locally. In order for a node $v$ to detect if it is happy, $v$ only has to ask all nodes $w \in S_v$ for their degrees.
2. In order for a node $v$ to perform $\mathsf{Exp}(X), X \subseteq N(v)$, $v$ must ask a subset of nodes $w \in S_v$ for their $S_{w \not\to r_w(v)}$, which is straightforward to implement.
3. When needed, a node $w$ can inform nodes $v \in S_v$ that they have been added to $S_w$. After which it is straightforward for $v$ to add $w$ to $S_v$.

In all of the routines above, it is possible for $v \in S_w$ for some $w$, such that $w \notin S_v$. This can happen when $v$ does not maintain a symmetric view towards a direction in fullDirs in Step 1(c). This can cause $> n^\delta$ nodes querying node $v$, breaking the communication constraint of the low-space MPC model. The following scheme resolves the issue. Let us first restate a tree structure that is useful to carry out computations on a set or on a collection of sets, in $O(1)$ low-space MPC rounds and with $O(n+m)$ global memory.

Denote the collection of machines we are using for the algorithm as $M = \{M_1, M_2, \ldots, M_l\}$. Let us allocate a new collection of empty machines $M' = \{M'_1, M'_2, \ldots, M'_l\}$. For every node $w \in S_v$ of a node $v$ hosted by $M_j$, send a directed edge $(v, w)$ to $M'_j$. Let us call all edges of form $(x, y)$ as the *outgoing* edges of $x$ and *incoming* edges of $y$. Along with the edge, send the address of machine $M_j$.

Define sets $A_1, \ldots, A_k$ such that set $A_i$ contains all incoming edges of node $i$. Apply Definition A.1 such that sets $A_1, \ldots, A_k$ are stored in $M'$ in lexicographically sorted order (by the ID of $i \in \{1, \ldots, k\}$ and within each set $A_i$, the elements are sorted increasingly). By Definition A.1, for each $i \in \{1, \ldots, k\}$ such that the elements of $A_i$ appear on at least 2 different machines, there is a tree of constant depth containing the machines that store elements of $A_i$ as leafs and where each inner node of the tree has at most $n^{\delta/2}$ children. Let us denote this kind of tree as $\mathcal{A}_i$.

Observe that every $\mathcal{A}_i$ corresponds to a node $i$ that has $> n^\delta$ incoming edges, which is exactly the problematic case we have set out to deal with. The root of $\mathcal{A}_i$ can ask for $S_i$ from the machine in $M$ hosting node $i$, and

distributes $S_i$ to all leaf nodes hosting the incoming edges (this requires a communication bandwidth of $O(n^{3\delta/2})$). We also establish a mapping from the machines in $M$ to machines in $M'$ such that the machine in $M$ hosting $u$ and $S_u$ knows the machines in $M'$ hosting edges $(u, v)$ for every $v \in S_u$. This is straightforward to implement, since when we distributed the edges to $M'$, we also distributed the corresponding addresses of machines in $M$.

Let us describe what effectively happens when a node $u$ asks for $S_{v \nrightarrow r_v(u)}$ of node $v \in S_u$ if $v$ has $> n^\delta$ incoming edges. The machine $M_u \in M$ hosting node $u$ queries the machine $M'_{(u,v)} \in M'$ for $S_{v \nrightarrow r_v(u)}$, where $M'_{(u,v)}$ is the machine hosting edge $(u, v)$. Due to the design of the aggregation tree, machine $M'_{(u,v)}$ is a leaf of tree $\mathcal{A}_v$ and has at most $n^\delta$ elements (edges) stored on it. The queries to machine $M'_{(u,v)}$ comprise an incoming message size of $O(n^\delta)$. Answering the queries would require a communication bandwidth of $O(n^{2\delta})$.

We can reduce the communication bandwidth of $O(n^{3\delta/2})$ (the distribution of $S_i$) and $O(n^{2\delta})$ (the leaves of $\mathcal{A}_v$ answering queries) to $O(n^\delta)$ by using $\delta/2$ instead of $\delta$ for the whole algorithm. □

## References


[1] Noga Alon, Lásló 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.

[2] Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. Parallel Graph Connectivity in Log Diameter Rounds. In *FOCS*, pages 674–685, 2018.

[3] Alkida Balliu, Sebastian Brandt, Yuval Efron, Juho Hirvonen, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Classification of Distributed Binary Labeling Problems. In *DISC*, 2020.

[4] Alkida Balliu, Sebastian Brandt, Manuela Fischer, Rustam Latypov, Yannic Maus, Dennis Olivetti, and Jara Uitto. Exponential Speedup over Locality in MPC with Optimal Memory. In Christian Scheideler, editor, *36th International Symposium on Distributed Computing (DISC 2022)*, volume 246 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:21, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[5] Alkida Balliu, Sebastian Brandt, Juho Hirvonen, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. Lower Bounds for Maximal Matchings and Maximal Independent Sets. *Journal of the ACM*, 68(5):39:1–39:30, 2021.

[6] Alkida Balliu, Sebastian Brandt, Fabian Kuhn, and Dennis Olivetti. Improved Distributed Lower Bounds for MIS and Bounded (Out-)Degree Dominating Sets in Trees. In *Proc. 40th ACM Symposium on Principles of Distributed Computing (PODC)*, 2021.

[7] Alkida Balliu, Sebastian Brandt, Fabian Kuhn, and Dennis Olivetti. Distributed Δ-coloring plays hide-and-seek. In *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing*, pages 464–477. ACM, 2022.

[8] Alkida Balliu, Sebastian Brandt, and Dennis Olivetti. Distributed Lower Bounds for Ruling Sets. *SIAM Journal on Computing*, 51(1):70–115, 2022.

[9] Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. Almost Global Problems in the LOCAL Model. In *DISC*, pages 9:1–9:16, 2018.

[10] Alkida Balliu, Keren Censor-Hillel, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Locally Checkable Labelings with Small Messages. In *the Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 8:1–8:18, 2021.

[11] Alkida Balliu, Mohsen Ghaffari, Fabian Kuhn, and Dennis Olivetti. Node and Edge Averaged Complexities of Local Graph Problems. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing (PODC)*, 2022.

[12] Alkida Balliu, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Dennis Olivetti, and Jukka Suomela. New Classes of Distributed Time Complexity. In *STOC*, pages 1307–1318, 2018.

[13] Philipp Bamberger, Fabian Kuhn, and Yannic Maus. Efficient Deterministic Distributed Coloring with Small Bandwidth. *CoRR*, abs/1912.02814, 2019.

[14] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, and Vahab Mirrokni. Near-Optimal Massively Parallel Graph Connectivity. In *FOCS*, 2019.

[15] Sebastian Brandt, Manuela Fischer, and Jara Uitto. Breaking the Linear-Memory Barrier in MPC: Fast MIS on Trees with Strongly Sublinear Memory. In *SIROCCO*, 2019.

[16] Sebastian Brandt, Christoph Grunau, and Vaclav Rozhon. The landscape of distributed complexities on trees and beyond. In *the Proceedings of the 41st Symposium on Principles of Distributed Computing (PODC)*, 2022.

[17] Sebastian Brandt, Rustam Latypov, and Jara Uitto. Brief Announcement: Memory Efficient Massively Parallel Algorithms for LCL Problems on Trees. In *35th International Symposium on Distributed Computing (DISC 2021)*, pages 50:1–50:4, 2021.

[18] Yi-Jun Chang. The Complexity Landscape of Distributed Locally Checkable Problems on Trees. In *DISC*, pages 18:1–18:17, 2020.
[19] Yi-Jun Chang, Manuela Fischer, Mohsen Ghaffari, Jara Uitto, and Yufan Zheng. The Complexity of $(\Delta+1)$-Coloring in Congested Clique, Massively Parallel Computation, and Centralized Local Computation. In *PODC*, 2019.
[20] Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An Exponential Separation between Randomized and Deterministic Complexity in the LOCAL Model. *SIAM J. Comput.*, 48(1):122–143, 2019.
[21] Yi-Jun Chang and Seth Pettie. A Time Hierarchy Theorem for the LOCAL Model. *SIAM J. Comput.*, 48(1):33–69, 2019.
[22] Sam Coy and Artur Czumaj. Deterministic Massively Parallel Connectivity. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2022.
[23] Artur Czumaj, Peter Davies, and Merav Parter. Graph Sparsification for Derandomizing Massively Parallel Computation with Low Space. In *the Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 175–185, 2020.
[24] Artur Czumaj, Peter Davies, and Merav Parter. Component Stability in Low-Space Massively Parallel Computation. In *PODC*, 2021.
[25] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, pages 107–113, 2008.
[26] Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. Conditional Hardness Results for Massively Parallel Computation from Distributed Lower Bounds. In *FOCS*, pages 1650–1663, 2019.
[27] Mohsen Ghaffari and Jara Uitto. Sparsifying Distributed Algorithms with Ramifications in Massively Parallel Computation and Centralized Local Computation. In *SODA*, 2019.
[28] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *ACM SIGOPS Operating Systems Review*, pages 59–72, 2007.
[29] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A Model of Computation for MapReduce. In *SODA*, 2010.
[30] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local Computation: Lower and Upper Bounds. *Journal of ACM*, 63:17:1–17:44, 2016.
[31] Christoph Lenzen and Roger Wattenhofer. Brief Announcement: Exponential Speed-Up of Local Algorithms Using Non-Local Communication. In *PODC*, 2010.
[32] Nathan Linial. Distributive Graph Algorithms – Global Solutions from Local Data. In *FOCS*, 1987.
[33] Nathan Linial. Locality in Distributed Graph Algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.
[34] Michael Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM Journal on Computing*, 15:1036–1053, 1986.
[35] Moni Naor and Larry Stockmeyer. What Can Be Computed Locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995.
[36] Tim Roughgarden, Sergei Vassilvitskii, and Joshua R. Wang. Shuffles and Circuits (On Lower Bounds for Modern Parallel Computation). *Journal of the ACM*, 2018.
[37] Salil P. Vadhan. Pseudorandomness. *Foundations and Trends in Theoretical Computer Science*, 7(1-3):1–336, 2012.
[38] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.
[39] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *the Proceedings of the SENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010.