On the Node-Averaged Complexity of Locally Checkable Problems on Trees[*]

Alkida Balliu · alkida.balliu@gssi.it · GSSI L'Aquila, Italy

Sebastian Brandt · brandt@cispa.de · CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

Fabian Kuhn · kuhn@cs.uni-freiburg.de · University of Freiburg, Germany

Dennis Olivetti · dennis.olivetti@gssi.it · GSSI L'Aquila, Italy

Gustav Schmid · schmidg@informatik.uni-freiburg.de · University of Freiburg, Germany

Over the past decade, a long line of research has investigated the distributed complexity landscape of locally checkable labeling (LCL) problems on bounded-degree graphs, culminating in an almost-complete classification on general graphs and a complete classification on trees. The latter states that, on bounded-degree trees, any LCL problem has deterministic *worst-case* time complexity $O(1)$, $\Theta(\log^* n)$, $\Theta(\log n)$, or $\Theta(n^{1/k})$ for some positive integer k, and all of those complexity classes are nonempty. Moreover, randomness helps only for (some) problems with deterministic worst-case complexity $\Theta(\log n)$, and if randomness helps (asymptotically), then it helps exponentially.

In this work, we study how many distributed rounds are needed *on average per node* in order to solve an LCL problem on trees. We obtain a partial classification of the deterministic *node-averaged* complexity landscape for LCL problems. As our main result, we show that every problem with worst-case round complexity $O(\log n)$ has deterministic node-averaged complexity $O(\log^* n)$. We further establish bounds on the node-averaged complexity of problems with worst-case complexity $\Theta(n^{1/k})$: we show that all these problems have node-averaged complexity $\widetilde{\Omega}(n^{1/(2^k-1)})$, and that this lower bound is tight for some problems.

1 Introduction

The family of locally checkable labeling (LCL) problems was introduced in the seminal work of Naor and Stockmeyer [29] and since then, understanding the distributed complexity of computing LCLs has been at the core of the research on distributed graph algorithms. Roughly speaking, LCLs are labelings of the nodes or edges of a graph $G = (V, E)$ with labels from a finite alphabet such that some local, constant-radius condition holds at all the nodes. In the distributed context, G represents a network and one typically assumes that the nodes of G can communicate over the edges of G in synchronous rounds. If this communication is unrestricted, this is known as the LOCAL model of computation and if messages must consist of $O(\log n)$ bits (where n is the number of nodes), it is known as the CONGEST model. In our paper, we focus on the LOCAL model and we therefore do

[*]This work has been partially funded by the European Union - NextGenerationEU under the Italian Ministry of University and Research (MUR) National Innovation Ecosystem grant ECS00000041 - VITALITY – CUP: D13C21000430001, and by the German Research Foundation (DFG), Grant 491819048.

not explicitly analyze the required message sizes of our algorithms. We however believe that all our algorithms can be made to work in the CONGEST model with minor modifications.

Often LCL problems are studied in the context of bounded-degree graphs. In this case, LCLs include problems such as properly coloring the nodes of G with $\Delta + 1$ colors, where Δ is the maximum degree of G. Especially over the last decade, researchers have obtained a thorough understanding of the complexity landscape of distributed LCL problems in general bounded-degree graphs [19, 18, 23, 30, 11, 7] and also in more special graph families such as in particular in bounded-degree trees [24, 12, 18, 19, 8, 16]. Most of this work focuses on the classic notion of worst-case complexity: If all nodes start a computation at time 0 and communicate in synchronous rounds, how many rounds are needed until *all nodes* have decided about their outputs. In some case however, the worst-case round complexity might be determined by a small number of nodes that require a lot of time to compute their outputs, while most of the nodes find their outputs much faster. Consider for example the simple randomized $(\Delta + 1)$-coloring algorithm where in every step, every node picks a random available color and permanently keeps this color if there is no conflict. It is not hard to show that in every step, every uncolored node becomes colored with constant probability [25]. Hence, while we need $\Omega(\log n)$ steps (and thus also $\Omega(\log n)$ rounds) until all nodes are colored, for each individual node, the expected time to become colored is constant and consequently the time that nodes need on average to become colored is also constant w.h.p. In some contexts (e.g., when considering the energy cost of a distributed algorithm), this average completion time per node is more meaningful than the worst-case completion time and consequently, researchers have recently showed interest in determining the *node-averaged* time complexity of distributed graph algorithms [22, 14, 21, 10]. In the present paper, we continue this work and we study the *node-averaged complexity of LCL problems in bounded-degree trees.* Before describing our contributions, we first briefly summarize some of the relevant previous work.

Previous results on node-averaged complexity. The first paper that explicitly considered the node-averaged complexity of distributed graph algorithms is by Feuilloley [22]. The paper mainly considers LCL problems on paths and cycles (i.e., on graphs of maximum degree 2). It is known that on paths and cycles, when considering the worst-case complexity of LCL problems, randomization does not help and the only complexities that exist are $O(1)$, $\Theta(\log^* n)$, and $\Theta(n)$ [29, 18, 19]. In [22], it is shown that for deterministic algorithms, the worst-case complexity and the node-averaged complexity of LCL problems on paths and cycles is the same. This for example implies that the classic $\Omega(\log^* n)$ lower bound of [27] for coloring cycles with a constant number of colors also applies to node-averaged complexity. While this is true for deterministic algorithms, it is also shown in [22] that the randomized node-averaged complexity of 3-coloring paths and cycles is constant. As sketched above and also explicitly proven in [14], the same is true for the more general problem of computing a $(\Delta + 1)$-coloring in arbitrary graphs. While the results of [22] imply results for general LCLs on paths and cycles, the additional work on node-averaged complexity focused on the complexity of specific graph problems, in particular on the complexity of well-studied classic problems such as computing a maximal independent set (MIS) or a vertex coloring of the given graph. Barenboim and Tzur [14] show that in graphs of small arboricity, some coloring problems have a deterministic node-averaged complexity that is significantly smaller than the corresponding worst-case complexity. For example, it is shown that if the arboricity is constant, an $O(k)$-vertex coloring can be computed in node-averaged complexity $O(\log^{(k)} n)$ for any fixed integer $k \geq 1$, where $O(\log^{(k)} n)$ denotes the k times iterated logarithm of n. As one of the main results of [10], it was shown that the MIS lower bound of [26] can be generalized to show that even with randomization, computing an MIS on general (unbounded degree) graphs requires node-averaged complexity $\Omega(\sqrt{\log n / \log\log n})$. Hence, while the problem of coloring with $(\Delta + 1)$ colors and, as also shown in [10], the problem of computing a 2-ruling set have randomized algorithms with constant node-averaged complexity, the same is not true for the problem of computing an MIS.

LCL complexity in bounded-degree trees. One of the goals of this paper is to make a step beyond understanding individual problems and to start studying the landscape of possible node-averaged complexities of general LCL problems. We do this by studying LCL problems on bounded-degree trees, a graph family that we believe is relevant and that has recently been studied intensively from a worst-case complexity point of view (e.g., [18, 19, 17, 12, 23, 30, 24]). In bounded-degree trees, we know that for deterministic algorithms, exactly the following worst-case complexities are possible: $O(1)$, $\Theta(\log^* n)$, $\Theta(\log n)$, and $\Theta(n^{1/k})$ for some fixed integer $k \geq 1$. It was shown in [24] (and earlier for a special subclass of LCLs in [12] and for paths in [29, 19]) that on bounded-degree trees, there are no deterministic or randomized asymptotically optimal algorithms with a time complexity in the range $\omega(1)$ to $o(\log^* n)$. Further, in [18], it was shown that even for general bounded-degree graphs, there are no deterministic LCL complexities in the range $\omega(\log^* n)$ to $o(\log n)$. Finally, it was shown in [19] that every LCL problem that requires $\omega(\log n)$ rounds on bounded-degree trees has a worst-case deterministic and randomized complexity of the form $\Theta(n^{1/k})$ for some fixed integer $k \geq 1$ (and all those complexities also exist). It is further known that randomization can only help for LCL problems with a deterministic complexity of $\Theta(\log n)$. Those problems have a randomized complexity of either $\Theta(\log n)$ or $\Theta(\log \log n)$ (and both cases exist) [19, 17]. We discuss additional related work on the complexity landscape of LCLs in Appendix A.

1.1 Our Contributions

As our main result, we show that the $\Theta(\log n)$ complexity class vanishes when considering the node-averaged complexity of LCLs on bounded-degree trees. More formally, we prove the following theorem.

Theorem 1. *Let Π be an LCL problem for which there is an $O(\log n)$-round deterministic algorithm on bounded-degree trees. Then, Π can be solved deterministically with node-averaged complexity $O(\log^* n)$ on bounded-degree trees.*

A standard example for an LCL problem that requires $\Theta(\log n)$ rounds deterministically is the problem of 3-coloring a tree. So for 3-coloring Theorem 1 states that there is a deterministic distributed 3-coloring algorithm, for bounded degree trees, with node-averaged complexity $O(\log^* n)$ rounds. Meaning that the average node terminates after $O(\log^* n)$ rounds. Note that for 3-coloring trees deterministically, this is tight. As shown in [22], 3-coloring has deterministic node-averaged complexity $\Omega(\log^* n)$ even on paths. Below, we will use the 3-coloring problem as a simple example to illustrate some of the challenges in obtaining the above theorem, but first we state the rest of our results.

In addition to Theorem 1, we also investigate the node-averaged complexity of LCL problems that require polynomial time in the worst case (i.e., time $\Theta(n^{1/k})$ for some integer $k \geq 1$). We show that for such problems, also the node-averaged complexity is polynomial. However at least in some cases, it is possible to obtain a node-averaged complexity that is significantly below the worst-case complexity. In [19], the hierarchical $2\frac{1}{2}$-coloring problem with parameter k is defined as an example problem with worst-case complexity $\Theta(n^{1/k})$. We show that the node-averaged complexity of this LCL problem is significantly smaller.

Theorem 2. *The deterministic node-averaged complexity of the hierarchical $2\frac{1}{2}$-coloring problem with parameter k is $O(n^{1/(2^k-1)})$.*

Finally, we show that for a problem with worst-case complexity $\Theta(n^{1/k})$, this is essentially the best possible node-averaged complexity. Meaning that we also prove that our algorithm for hierarchical $2\frac{1}{2}$-coloring problems is optimal up to one $\log n$ factor.

Theorem 3. *Let Π be an LCL problem with (deterministic or randomized) worst-case complexity $\Omega(n^{1/k})$. Then, the randomized node-averaged complexity of Π is $\Omega(n^{1/(2^k-1)}/\log n)$.*

Note that the algorithm of Theorem 2 is deterministic, and that the lower bound of Theorem 3 holds for randomized algorithms as well.

1.2 High-level Ideas and Challenges

We next discuss some of the ideas that lead to the known results about solving LCL problems on bounded-degree trees and we highlight some of the challenges that one has to overcome and some of the ideas we use to prove Theorems 1 to 3.

Rake-and-compress decomposition. We start by sketching a generic algorithm that can be used to solve all LCL problems in bounded-degree trees. The generic algorithm can be used to obtain algorithms with an asymptotically optimal worst-case complexity for all problems with worst-case complexity $\Omega(\log n)$. As a first step, the algorithm uses a technique that is known as *rake-and-compress* [28] to partition the nodes of a given tree $T = (V, E)$ into $O(\log n)$ layers such that each layer is either a rake layer that consists of a set of independent nodes or it is a compress layer that consists of a sufficiently separated set of paths. Every node in a rake layer has at most one neighbor in a higher layer, and in each path of a compress layer, the two nodes at the end have exactly one neighbor in a higher layer and the other nodes on the path have no neighbors in a higher layer.[1] Such a decomposition can be computed in an iterative process that produces the layers in increasing order. Given some tree (or forest), a rake layer can be obtained by taking the set of all leaf nodes[2] and a compress layer can be created by the paths (or more precicely by the inner part of the paths) induced by degree-2 nodes. It is not hard to show that when alternating rake and compress layers, this process completes after creating $O(\log n)$ layers [28].

Applying the decomposition. As an example of how to use rake-and-compress to solve an LCL problem, we look at the case of 3-coloring the nodes of a tree T. Given a decomposition into rake and compress layers, this can be done in $O(\log n)$ rounds as follows. First, color each of the paths of the compress layers with $O(1)$ colors. This can be done in $O(\log^* n)$ rounds. Then, the 3-coloring of T is computed by starting at the highest layer of the decomposition. When processing a rake layer, each node can just be colored with a color different from its (at most one) neighbor in a higher layer. When processing a compress layer, we just have to 3-color the paths of the layer such that each node at the end of a path picks a color that differs from the color of its neighbor in a higher layer. Given the initial $O(1)$-coloring of the path, this can be done in constant time for each path. The time to compute the coloring is therefore proportional to the number of layers and thus $O(\log n)$. The generic algorithm for solving more general LCL problems is more involved, but still similar at a high level. While creating the decomposition, for each node v, one can create a list of labels that can be assigned to v such that the labeling of lower layer nodes that depend on v can still be completed. The LCL problem needs to allow labelings that are flexible enough such that when having long paths of nodes that each can be the root of an arbitrary subtree, the nodes of the path can still be labeled efficiently (in constant time given an appropriate initial coloring of the path).

[1]The actual decomposition that we use is a bit more complicated and the formal definition (see Definition 6) requires some additional details.

[2]When two degree-1 nodes are neigbors, one just takes one of the two nodes.

Implementation with low node-averaged complexity. The main challenge when trying to obtain an $o(\log n)$ node-averaged complexity is the following. The generic algorithm first computes the decomposition and it then computes the labeling by starting with the nodes in the highest layers. In the worst case, we therefore need $\Theta(\log n)$ rounds before even the label of a single node is determined. And moreover, most of the nodes are in the first few layers, which are labeled at the very end of the algorithm. In order to obtain a low node-averaged complexity, we therefore need to label most of the nodes already in the "bottom-up" phase when creating the rake and compress layers. For some problems, this is challenging: for example, in the 3-coloring problem, if we ever obtain a node with 3 neighbors of lower layers that have 3 different colors, then we cannot complete the solution in any valid way. Hence, we have to label the nodes in such a way that the "top-down" phase is still able to extend the partial labeling to a valid labeling of all the nodes. To keep things simple, we here assume that the tree has diameter $O(\log n)$. In this case, it suffices to create rake layers and we do not need compress layers. We further only look at the problem of 3-coloring the nodes of T. This problem is significantly easier to handle than general $O(\log n)$-worst case complexity LCL problems, the solution for 3-coloring however already requires some of the ideas that we use in the general case.

Let us therefore assume that we have an $O(\log n)$-diameter tree T with maximum degree $\Delta = O(1)$. If we decompose by using only rake layers, we obtain $O(\log n)$ layers, where each layer is an independent set of T and every node has exactly one neighbor in a higher layer, except for the single node u that is in the top layer. We refer to u as the root node and for each node v, we refer to the single neighbor w of v in a higher layer as the parent of v. Note that when assigning a color to a node v in the top-down phase, only v's parent has already been assigned a color. To complete the top-down phase, it therefore suffices if every node v can choose its color from an arbitrary subset S_v of size 2 of the colors. So if no nodes below v have already decided on a color, this will always be possible. Hence, if we try to color some nodes already in the bottom-up phase, we have to make sure that all the uncolored nodes still have at least two available colors. This is guaranteed as long as every uncolored node has at most one colored neighbor.

When constructing the layering we therefore proceed as follows. We only color nodes that have already been assigned to some rake layer. Whenever we decide to color a node v in the bottom-up phase, we also directly color the whole subtree of v.[3] The high-level idea of the algorithm to achieve this is as follows. After each rake step, i.e., after each creation of a new layer, we check whether or not there are some nodes that can be colored. Consider the situation after the t^{th} rake step, let $G^{(t)}$ be the set of nodes that have not been raked at that time (i.e., that have not been assigned to some layer), and let $R^{(t)}$ be the set of nodes that have already been assigned to some layer. Note that if a node $u \in G^{(t)}$ has some neighbor $v \in R^{(t)}$, then u will in the end be the unique neighbor of v in a higher layer. We can therefore think of the nodes in $G^{(t)}$ as the roots of the already raked subtrees. This is illustrated in Figure 1. After each rake step t, each node $u \in G^{(t)}$ tries to color some node at distance 2 in its subtree.[4] Node u chooses v^* to be a node at distance 2 in its subtree such that the subtree rooted at v^* has the largest number of uncolored nodes among all nodes at distance 2 of u in the subtree of u (observe that nodes can keep track of such numbers). If there are no colored 2-hop neigbors of v^* outside the subtree of v^* (i.e., no colored siblings of v^*), then u decides to color v^* and its complete subtree. Otherwise, no new nodes in u's subtree are getting colored. If v^* and its subtree get colored, then a constant fraction of the uncolored nodes in u's subtree get colored. Otherwise, a sibling v' of v^* with a larger subtree has already been colored while u was the root of

[3]After coloring the root of a subtree, the coloring of the subtree can be done in parallel while proceedings with the rest of the algorithm.

[4]By only coloring nodes at distance at least 2 from u, we make sure that neighbors of nodes that are not yet layered remain uncolored.

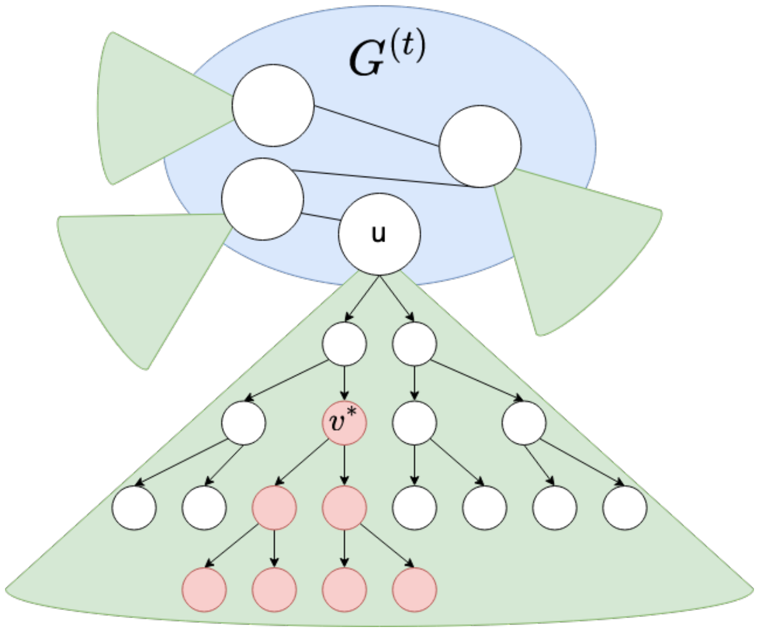

Figure 1: The graph $G^{(t)}$ of nodes that are not yet raked away is colored blue. The already raked away nodes $R^{(t)}$ are colored green. The node u chooses v^* since it has the largest subtree, colored in red, attached and both v^* as well as its entire subtree become colored.

the tree. Note that at this time, the subtree of v^* was already in the same state and therefore v' colored more nodes than v^* does. One can use this to show that whenever the height of a raked subtree increases, a constant fraction of the uncolored nodes gets colored. One can further show that this suffices to show that over the whole tree, a constant fraction of the remaining nodes gets colored every constant number of rounds and thus the node-averaged complexity is constant. The algorithm and the analysis for the general family of LCLs for which Theorem 1 holds uses similar basic ideas, dealing with the general case is however significantly more involved.

Improved upper bounds in the polynomial regime. We prove that the node-averaged complexity of the hierarchical $2\frac{1}{2}$-coloring problem with parameter k is $O(n^{1/(2^k-1)})$. In order to give some intuition for this, we focus on the case $k=2$ where the worst-case complexity is $\Theta(\sqrt{n})$. Instead of providing a formal definition of the problem, it is helpful to present the problem by describing how a worst-case instance for the problem looks like, and how a solution in such an instance looks like. A worst-case instance for this problem consists of a path P of length $\Theta(\sqrt{n})$, where to each node v_j of P is attached a path Q_j of length $\Theta(\sqrt{n})$. We call the nodes of the path P *p-nodes* and we call the nodes of a path Q_j *q-nodes*. For each path Q_j, the algorithm has to decide to either 2-color it or to mark the whole path as *decline*. Then, the subpaths of P induced by nodes that are neighbors of q-nodes that output decline need to be labeled with a proper 2-coloring. In particular, *decline* is not allowed on p-nodes. Let us now describe an algorithm with optimal worst-case complexity for instances with a similar structure, but where the paths may have different lengths. For q-nodes, the algorithm first checks if the length of the path containing those nodes is $O(\sqrt{n})$ (note that, in order to perform this operation, the algorithm needs to know n, and it is actually unknown whether an LCL problem can have $\Theta(\sqrt{n})$ worst-case complexity when n is unknown). In such a case, the algorithm is able to produce a proper 2-coloring of the path. Otherwise, the path is marked as decline. Then, it is possible to prove that the subpaths of P induced by nodes having q-node neighbors that output decline must be of length $O(\sqrt{n})$, and hence they can be properly 2-colored in $O(\sqrt{n})$ rounds. We observe that in the worst-case instance described above, the majority of the nodes of the graph are q-nodes, and hence, from an average point of view, it would be fine if p-nodes spend more time. In fact, it is possible to improve the node-averaged complexity of the described algorithm by letting q-nodes run for at most $O(n^{1/3})$ rounds and p-nodes for at most $O(n^{2/3})$ rounds. In this case, a worst-case instance contains a path P of length $O(n^{2/3})$ and all paths Q_j are of length $O(n^{1/3})$. We obtain that both the p-nodes and the q-nodes contribute $O(n^{4/3})$ to the sum of the running times, obtaining a node-averaged complexity of $O(n^{1/3})$.

Lower bounds in the polynomial regime. It is known by [16] that if an LCL problem Π has worst-case complexity $o(n^{1/k})$, then it can actually be solved in $O(n^{1/(k+1)})$ rounds. The intuition about what determines the exact value of k in the complexity of a problem is related to how many compress layers of a rake-and-compress decomposition one can handle. In the example presented above, namely 3-coloring, one can handle an arbitrary number of compress paths and that is the reason why the problem can be solved in $O(\log n)$ rounds. In particular, no matter how many rake or compress operations have been applied, we can handle any compress path by producing a 3-coloring on it and leaving the endpoints uncolored (such nodes can decide their color after their higher layer neighbors picked a color), and this can be done fast. Not all problems are of this form, that is, for some problems we cannot handle an arbitrary amount of compress paths: it is possible to define problems in which different labels need to be used in compress paths of different layers (hierarchical $2\frac{1}{2}$ coloring is indeed such a problem where in fact p-nodes are not allowed to output *decline*). For such problems, it may not be possible at all to efficiently produce a valid labeling for long compress paths of layers that are too high, say of layers strictly more than k. In order to solve this issue, we can modify the generic algorithm sketched above by increasing the number of rake operations that are performed between each pair of compress operations. When using $\Omega(n^{1/(k+1)})$ rake operations at the beginning and between any two compress operations, the total number of compress layers is at most k. This however makes the algorithm slower, resulting in a complexity of $\Theta(n^{1/(k+1)})$ (while 3-coloring has worst-case complexity $\Theta(\log n)$).

In other words, for some LCL problems, compress paths are something that is difficult to handle, and the number of compress layers that we can recursively handle is what determines the complexity of a problem. If we can handle an arbitrary amount of compress layers, then the problem can be solved in $O(\log n)$ rounds, but if we can handle only a constant amount of compress layers, say k, then the complexity of the problem is $\Theta(n^{1/(k+1)})$. In [16] it is proved that, if a problem has complexity $o(n^{1/k})$, then it is possible to handle k compress layers, implying a complexity of $O(n^{1/(k+1)})$. We show that the same can be obtained by starting from an algorithm $\mathcal{A}$ with node-averaged complexity $o(n^{1/(2^k-1)}/\log n)$, implying that if a problem has complexity $\Omega(n^{1/k})$, then it cannot have node-averaged complexity $o(n^{1/(2^k-1)}/\log n)$, since otherwise it would imply that the problem can actually be solved in $O(n^{1/(k+1)})$ rounds in the worst case, which then leads to a contradiction. Starting from an algorithm that only has guarantees on its node-averaged complexity instead of on its worst-case complexity introduces many additional challenges that we need to tackle. For example, in [16] it is argued that an $o(n^{1/k})$-rounds algorithm can never see both the endpoints of a carefully crafted path that is too long. This kind of reasoning, that is very common when we deal with worst-case complexity, does not work for node-averaged complexity.

2 Road Map

The remainder of the paper is organized as follows.

Preliminaries. We start, in Section 3, by providing some definitions. In particular, we define the class of problems that we consider, and the notion of node-averaged complexity.

Locally checkable labelings. We continue, in Section 4, by providing an overview of what is known about solving LCL problems on trees of bounded degree. We present a generic algorithm that is known to be able to solve all LCLs, that has optimal worst-case complexity whenever the considered problem has worst-case complexity $\Omega(\log n)$. The content of this section is heavily based on existing results.

A fast algorithm for problems with intermediate complexity. In Section 5, we present an algorithm with node-averaged complexity $O(\log^* n)$, that is able to solve all problems that have $O(\log n)$ worst-case complexity. This algorithm is based on the one presented in Section 4, but we need to tackle many challenges in order to improve its node-averaged complexity.

A fast algorithm for some problems with polynomial complexity. In Section 6 we consider a class of problems, called hierarchical $2\frac{1}{2}$ coloring, that are parametrized by an integer k. For these problems it is known that their worst-case complexity is $\Theta(n^{1/k})$. We show that these problems can be solved with node-averaged complexity $O(n^{1/(2^k-1)})$.

A lower bounds for problems with polynomial complexity. It is known that all LCL problems on trees either have worst-case complexity $O(\log n)$, or $\Theta(n^{1/k})$ for some integer $k \geq 1$. While we show in Section 5 that the worst-case complexity class $O(\log n)$ becomes $O(\log^* n)$ for node-averaged complexity, in Section 7 we show that all problems that have polynomial worst-case complexity also have polynomial node-averaged complexity. In particular, we show that if a problem has worst-case complexity $\Theta(n^{1/k})$, then it has node-averaged complexity $\Omega(n^{1/(2^k-1)}/\log n)$.

Open questions In Section 8, we provide some open questions.

More on LCLs In Appendix A, we provide additional related work about LCLs.

An algorithm for solving all LCLs in $O(D)$ rounds. In Appendix B, we show a simple bandwidth-efficient algorithm that solves all LCL problems in $O(D)$ rounds, where D is the diameter of the graph, and then we give some intuition on the challenges that one needs to tackle in order to improve its complexity. The content of this section can be useful to better understand Section 4.

Different ways to define LCLs. In Appendix C, we provide different definitions of LCLs, and we prove that the notion that we study (called black-white formalism) is equivalent, for node-averaged complexity, to the standard one studied in the literature (this was previously known only for the case of worst-case complexity).

3 Preliminaries

LCLs in the black-white formalism. We start by defining the class of problems that we consider, called LCLs in the black-white formalism. We show in Lemma 45 that on trees, studying this class of problems is equivalent to studying LCLs as they are usually defined in the literature. A problem Π described in the black-white formalism is a tuple $(\Sigma_{\text{in}}, \Sigma_{\text{out}}, C_W, C_B)$, where:

- Σ_{in} and Σ_{out} are finite sets of labels.
- C_W and C_B are both multisets of pairs, where each pair $(\ell_{\text{in}}, \ell_{\text{out}})$ is in $\Sigma_{\text{in}} \times \Sigma_{\text{out}}$.

Solving a problem Π on a graph G means that:

- $G = (W \cup B, E)$ is a graph that is properly 2-colored, and in particular each node $v \in W$ is labeled $c(v) = W$, and each node $v \in B$ is labeled $c(v) = B$.
- To each edge $e \in E$ is assigned a label $i(e) \in \Sigma_{\text{in}}$.
- The task is to assign a label $o(e) \in \Sigma_{\text{out}}$ to each edge $e \in E$ such that, for each node $v \in W$ (resp. $v \in B$) it holds that the multiset of incident input-output pairs is in C_W (resp. in C_B).

Note that when expressing a given LCL problem on a tree T in the black-white formalism, we often have to modify the tree T as follows. We subdivide every edge e of T by inserting one node in the middle of the edge. Each edge is then split into two "half-edges" and the new tree is trivially properly 2-colored (say the original nodes of T are the black nodes and the newly inserted nodes for each edge of T are the white nodes).

Node-averaged complexity. We define the notion of node-averaged complexity as in [10]. Let $\mathcal{A}$ be an algorithm that solves a problem Π. Assume $\mathcal{A}$ is run on a given graph $G = (V, E)$. Let $v \in V$. We define $T_v^G(\mathcal{A})$ to be the number of rounds after which v terminates when running $\mathcal{A}$. The node-averaged complexity of an algorithm $\mathcal{A}$ on a family of graphs $\mathcal{G}$ is defined as follows.

$$\mathsf{AVG}_V(\mathcal{A}) := \max_{G \in \mathcal{G}} \frac{1}{|V|} \cdot \mathbb{E}\left[\sum_{v \in V(G)} T_v^G(\mathcal{A})\right] = \max_{G \in \mathcal{G}} \frac{1}{|V|} \cdot \sum_{v \in V(G)} \mathbb{E}\left[T_v^G(\mathcal{A})\right]$$

The complexity of Π is defined as the lowest complexity of all the algorithms that solve Π.

4 Locally Checkable Labelings

In this section we introduce a class of problems called Locally Checkable Labelings (LCLs) and we summarize known results about the worst-case complexity of LCLs in the case in which we restrict the family of considered graphs to be trees of bounded degree. We first provide an overview of what are the possible complexities of LCLs on trees, and then we describe a generic method that can be used to solve some of these problems optimally.

4.1 Introduction

LCLs have been extensively studied on trees of bounded degree and we know that in this family of graphs, these problems can only have the deterministic complexities $O(1)$, $\Theta(\log^* n)$, $\Theta(\log n)$, and $\Theta(n^{1/k})$ for any $k \in \mathbb{N}$ [19, 16, 8, 6, 3]. Moreover, we know that randomness may only help for problems with deterministic complexity $\Theta(\log n)$. Finally, we know that for problems with deterministic complexity $\Omega(\log n)$, given the description of an LCL problem, we can automatically determine its time complexity. These decidability results have first been shown in two important papers [19, 16], which we summarize in the rest of the section.

On a high level, these decidability results have been proven as follows: first, there is a generic method to solve all problems, based on a procedure called *rake-and-compress*; then, it is shown that this method has optimal time complexity, meaning that if this method is not able to provide a fast algorithm, then the problem cannot be solved fast with any other algorithm. In Appendix B, we present the procedure for a simplified setting in which we aim at solving a restricted set of problems in $O(D)$ rounds. There, we also explain which challenges one needs to tackle in order to improve the running time.

4.2 A Generic Way to Solve All LCLs

In this section we present known results about solvability of LCLs. The content of this section is heavily based on results presented in [19, 16, 9], that provide a generic method that is able to solve any (solvable) LCL. This method has an optimal worst-case complexity for all problems that require $\Omega(\log n)$ worst-case rounds in the deterministic setting. In later sections, we will use some ingredients that we present in this section, in order to provide algorithms that have faster node-averaged complexity. We follow a similar route of [9]: in order to keep our proofs more accessible we prove our statements for LCLs expressed in the black-white formalism (which is simpler to deal with than standard LCLs as they are usually defined in the literature). In Lemma 45 we prove that on trees, for any standard LCL, we can define an LCL in the black-white formalism that has the same asymptotic node-averaged complexity as the original one, implying that our results hold for all standard LCLs as well.

Classes. A summary of the generic algorithm for solving any LCL in $O(D)$ rounds presented in Appendix B is the following: nodes of degree 1 are recursively removed from the tree; at each step, nodes that become leaves compute the set of labels that can be put on the edge connecting them with the rest of the tree, in a way that the labeling in their removed subtree can be completed in a valid manner; once all edges get a set assigned, it is possible to pick a valid labeling for all the edges by processing nodes in reverse order.

The set computed by a node essentially behaves as an interface between the remaining tree and the part of the tree that got removed, in the sense that it is not important what is the actual subtree, and the only thing that matters is the content of the set. Informally, we call this set the *class* of the subtree. In order to obtain algorithms that are faster than $O(D)$ rounds, in the removal process it is required to handle also nodes of degree 2, and this is what complicates the formal definition of class.

While the following definition is generic (in order to minimize repetition), it may help the reader to check Figure 2, which shows the two possible cases in which the definition will be applied.

Definition 4 ([9])**.** Assume we are given an LCL $\Pi = (\Sigma_{\text{in}}, \Sigma_{\text{out}}, C_W, C_B)$ in the black-white formalism. Consider a tree $G = (V, E)$, and a connected subtree $H = (V_H, E_H)$ of G. Assume that the edges connecting nodes in V_H to nodes in $V \setminus V_H$ are split into two parts, F_{incoming} and F_{outgoing}, that are called, respectively, the set of incoming and outgoing edges. Assume also that for each edge $e \in F_{\text{incoming}}$ is assigned a set $L_e \subseteq \Sigma_{\text{out}}$. This set is called *the label-set of e*. Let $\mathcal{L}_{\text{incoming}} = (L_e)_{e \in F_{\text{incoming}}}$. A *feasible labeling* of H w.r.t. F_{incoming}, F_{outgoing}, and $\mathcal{L}_{\text{incoming}}$ is a tuple $(L_{\text{outgoing}}, L_{\text{incoming}}, L_{\text{H}})$ where:

- L_{incoming} is a labeling $(l_e)_{e \in F_{\text{incoming}}}$ of F_{incoming} satisfying $l_e \in (\mathcal{L}_{\text{incoming}})_e$ for all $e \in F_{\text{incoming}}$,
- L_{outgoing} is a labeling $(l_e)_{e \in F_{\text{outgoing}}}$ of F_{outgoing} satisfying $l_e \in \Sigma_{\text{out}}$ for all $e \in F_{\text{outgoing}}$,
- L_H is a labeling $(l_e)_{e \in E_H}$ of E_H satisfying $l_e \in \Sigma_{\text{out}}$ for all $e \in E_H$,
- the output labeling of the edges incident to nodes of H given by $L_{\text{outgoing}}, L_{\text{incoming}}$, and L_H is such that all node constraints of each node $v \in V_H$ are satisfied.

Also, we define the following:

- a *class* is a set of feasible labelings,
- a *maximal class* is the unique inclusion maximal class, that is, it is the set of all feasible labelings,

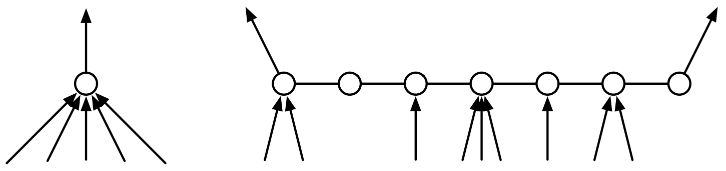

Figure 2: The figure illustrates the two cases of the label-set computation, where it is assumed that the incoming edges have already a label-set assigned and the goal is to assign a label-set to the outgoing edges; the left side depicts the case of a single node, the right side shows the case of a short path.

- an *independent class* is a class A such that for any $\left(L_{\text{outgoing}}, L_{\text{incoming}}, L_H\right) \in A$ and $\left(L'_{\text{outgoing}}, L'_{\text{incoming}}, L'_H\right) \in A$ the following holds. Let L''_{outgoing} be an arbitrary combination of L_{outgoing} and L'_{outgoing}, that is, $L''_{\text{outgoing}} = (l_e)_{e \in F_{\text{outgoing}}}$ where $l_e \in \{(L_{\text{outgoing}})_e, (L'_{\text{outgoing}})_e\}$. There must exist some L''_{incoming} and L''_H satisfying $\left(L''_{\text{outgoing}}, L''_{\text{incoming}}, L''_H\right) \in A$.

Note that the maximal class with regard to some given $\Pi, H, F_{\text{incoming}}, F_{\text{outgoing}}$, and $\mathcal{L}_{\text{incoming}}$, is unique. In contrast, there may be different ways (or none) to restrict a maximal class to a (nonempty) independent class.

Computing label-sets. We will use Definition 4 for two specific types of graphs H: either single nodes, or short paths. In each of these cases we will need to compute a label-set for each outgoing edge. We now describe the two cases in detail, and we provide a way to compute the label-sets. The cases are shown in Figure 2.

Definition 5 (label-set computation)**.** Assume we are given some function $f_{\Pi,k}$ (to be specified later). We define a function $g(v)$ that can be used to compute label-sets for the outgoing edges as a function of H, Π, F_{incoming}, F_{outgoing}, $\mathcal{L}_{\text{incoming}}$, and $f_{\Pi,k}$, for two specific types of graphs H.

- **Single nodes:** the graph H consists of a single node v that has a single outgoing edge e, and hence $F_{\text{outgoing}} = \{e\}$. All the other edges (which might be 0) are incoming, and for each of them we are given a label-set (where $\mathcal{L}_{\text{incoming}}$ represents this assignment). We assign, to the outgoing edge, the label-set $g(v)$, that consists of the set of labels that we can assign to the outgoing edge, such that we can pick a label for each incoming edge in a valid manner. More in detail, let B be the maximal class of H w.r.t. $\Pi, F_{\text{incoming}}, F_{\text{outgoing}}$, and $\mathcal{L}_{\text{incoming}}$. Then, we denote $g(v) = \bigcup_{(L_{\text{outgoing}}, L_{\text{incoming}}, L_H) \in B} \{(L_{\text{outgoing}})_e\}$. We have $g(v) \subseteq \Sigma_{\text{out}}$. Observe that each node v can compute $g(v)$ if it is given the value of $g(u)$ (that is, the label-set of the edge $\{u, v\}$) for each incoming edge $\{v, u\}$.

- **Short paths:** the graph H is a path of length between ℓ and 2ℓ, for some $\ell = O(1)$ that depends solely on Π and the target running time. The endpoints of the path are v_1 and v_2. The outgoing edges are $F_{\text{outgoing}} = \{e_1, e_2\}$, where e_1 (resp. e_2) is the outgoing edge incident to v_1 (resp. v_2). Let B be the maximal class of H. We assume to be given a function $f_{\Pi,k}$, that depends solely on Π and some parameter k (that, in turn, depends on the target running time), that maps a class B into an independent class $B' = f_{\Pi,k}(B)$. For $i \in \{1, 2\}$, let $g(v_i) = \bigcup_{(L_{\text{outgoing}}, L_{\text{incoming}}, L_H) \in B'} \{(L_{\text{outgoing}})_{e_i}\}$. We have $g(v_i) \subseteq \Sigma_{\text{out}}$. The label-set of e_1 (resp. e_2) is $g(v_1)$ (resp. $g(v_2)$). Observe that the values of $g(v_i)$, for $i \in \{1, 2\}$, can be computed given H and $\mathcal{L}_{\text{incoming}}$.

Tree decompositions. All problems with worst-case complexity $O(\log n)$ or $O(n^{1/k})$ for any $k \in \mathbb{N}$ can be solved by following a generic algorithm [19, 16, 9]. This algorithm decomposes the tree into layers by iteratively removing nodes in a rake-and-compress manner [28] (and then uses the computed decomposition to solve the given problem).

We first define the decomposition that the algorithm uses and then elaborate on how fast (and how) it can be computed.

Definition 6 ((γ, ℓ, L)-decomposition)**.** Given three integers γ, ℓ, L, a (γ, ℓ, L)-*decomposition* is a partition of $V(G)$ into $2L-1$ layers $V_1^R = (V_{1,1}^R, \ldots, V_{1,\gamma}^R), \ldots, V_L^R = (V_{L,1}^R, \ldots, V_{L,\gamma}^R)$, $V_1^C, \ldots, V_{L-1}^C$ such that the following hold.

1. Compress layers: The connected components of each $G[V_i^C]$ are paths of length in $[\ell, 2\ell]$, the endpoints have exactly one neighbor in a higher layer, and all other nodes do not have any neighbor in a higher layer.

2. Rake layers: The diameter of the connected components in $G[V_i^R]$ is $O(\gamma)$, and for each connected component at most one node has a neighbor in a higher layer.

3. The connected components of each sublayer $G[V_{i,j}^R]$ consist of isolated nodes. Each node in a sublayer $V_{i,j}^R$ has at most one neighbor in a higher layer or sublayer.

In the following, for completeness, we provide an algorithm of [19, 16] that shows how to compute a (γ, ℓ, L)-decomposition where, at the end, each node knows the layer that it belongs to.

1. Iteratively do the following, until the obtained graph is empty, starting with $i = 1$:

   (a) Iteratively, perform γ *rake operations.* For each $1 \leq j \leq \gamma$, the jth rake operation consists of removing all nodes that have degree 1 in the graph induced by the remaining nodes and putting them into the preliminary set $W_{i,j}^R$. For technical reasons, if two adjacent nodes have degree 1, the rake operation only removes one of them, chosen arbitrarily.

   (b) Perform one *compress operation*, that is, remove all nodes of degree 2 that, in the graph induced by the remaining nodes, are contained in paths of length at least ℓ where each node of the path has degree exactly 2. Put the removed nodes into the preliminary set W_i^C.

   (c) $i \leftarrow i + 1$.

2. Observe that, for all i, W_i^C is a collection of paths of length at least ℓ. Split long paths into shorter paths as follows. Compute, for each W_i^C, an independent set $I_i \subset W_i^C$ of nodes that satisfies two properties: no endpoint of any path in W_i^C is contained in I_i, and the maximal connected components of the graph obtained by removing the nodes in I_i from W_i^C are paths of length between ℓ and 2ℓ. For each i, we promote the nodes in I_i to the next rake layer, that is, we define V_i^C as $W_i^C \setminus I_i$, V_{i+1}^R as $W_{i+1}^R \cup I_i$, and $V_{i+1,1}^R$ as $W_{i+1,1}^R \cup I_i$. We denote with L the largest index i such that $V_i^R \cup V_{i-1}^C \neq \emptyset$.

The following lemma provides upper bounds for the deterministic worst-case complexity of computing a (γ, ℓ, L)-decomposition using the algorithm of [19, 16].

Lemma 7 ([19, 16])**.** *Assume $\ell = O(1)$. Then the following hold.*

- *For any positive integer k and $\gamma = n^{1/k}(\ell/2)^{1-1/k}$, a (γ, ℓ, k)-decomposition can be computed in $O(k \cdot n^{1/k})$ rounds.*

- *For $\gamma = 1$ and $L = O(\log n)$, a (γ, ℓ, L)-decomposition can be computed in $O(\log n)$ rounds.*

The generic algorithm with optimal worst-case complexity. We now explain the algorithm due to [19, 16, 9] that is able to solve any LCL Π with worst-case complexity $\Theta(\log n)$ or $\Theta(n^{1/k})$ asymptotically optimally. The initial objective is to compute a (γ, ℓ, L)-decomposition; however, first we need to determine suitable parameters γ, ℓ, and L. To this end, we begin by determining the worst-case time complexity of the given problem Π, which, by [19, 16], can be computed in finite time solely as a function of Π. Then, as a function of the target time complexity, we can determine γ and ℓ: if the target time complexity is $O(\log n)$, then $\gamma = 1$, and then ℓ can be computed as a function of Π; otherwise, if the target time complexity is $O(n^{1/k})$, then we can first compute ℓ as a function of Π and k, and then set $\gamma = n^{1/k}(\ell/2)^{1-1/k}$. By Lemma 7, we get that if in the former case we set $L = k$, and in the latter case we set $L = O(\log n)$, then a (γ, ℓ, L)-decomposition can be computed within a running time that matches the target complexity. In [19, 16] it is shown how to determine the value of ℓ in each case.

After computing a (γ, ℓ, L)-decomposition with the determined parameters, the decomposition is used to propagate label-sets up through the layers. The goal is to iteratively assign *label-sets* to edges, in a way that, when we handle nodes in layer i, all their edges connected to nodes of layers $< i$ have already a label-set assigned. As a last step of the generic algorithm, we will use the label-sets to pick a valid solution, by propagating labels through the layers in reverse order.

We first explain the procedure that computes a label-set for all edges of the graph by going up through the layers of the computed (γ, ℓ, L)-decomposition. This procedure uses Definition 5, and hence it requires to be given a function $f_{\Pi,k}$ (if the target runtime is $O(\log n)$, then let $k = \infty$). This function, as shown in [19, 16, 9], can be computed solely as a function of Π and the target time complexity (also, similarly as for ℓ, this function does not depend on n).

- For $j = 1, \ldots, \gamma$, by Definition 6, we have that the graph induced by $V^R_{i,j}$ is composed of isolated nodes. Each node $v \in V^R_{i,j}$ waits until all its neighbors z in lower layers have assigned a label-set to the edge $\{v, z\}$. We are now in the first case of Definition 5. Hence, v computes $g(v)$, the label-set of the edge connecting v to its only neighbor w in upper layers, if it exists, and sends this label-set to w.

- Nodes in V^C_i, by Definition 6, form paths of length between ℓ and 2ℓ. For each of these paths, nodes wait until a label-set has been computed for each edge connecting them to lower layers. We are now in the second case of Definition 5. Hence, the endpoints of the path can compute the label-sets for the edges connecting them to their neighbor in upper layers. Each endpoint sends its computed label-set to its higher-layer neighbor.

Now we explain how, in the last step of the generic algorithm, we use the computed label-sets to determine the final output labels by going through the layers of the computed decomposition in reverse order (i.e., from larger to smaller index). Observe that it may happen that rake nodes do not have any outgoing edge. In this case, a node considers its maximal class B (that is guaranteed to be non-empty), takes an arbitrary element $((), L_{\text{incoming}}, ())$ from B, and uses L_{incoming} to assign a label to each incoming edge. Then, we process the rest of the graph as follows.

- Each rake node, once a label l for its outgoing edge is assigned, considers its maximal class B, picks an arbitrary element $(L_{\text{outgoing}}, L_{\text{incoming}}, ())$ compatible with l from B, and uses L_{incoming} to assign a label to each incoming edge, where compatible with l means that L_{outgoing} assigns l to the outgoing edge.

- Similarly, each short path, once a label is assigned to the edges outgoing from the endpoints, can pick a valid assignment for each internal and incoming edge.

This concludes the description of the generic algorithm. We note that the generic algorithm does not require a specific (γ, ℓ, L)-decomposition—any (γ, ℓ, L)-decomposition (for the parameters γ, ℓ, L determined in the beginning of the generic algorithm) works. We will make use of this fact when designing algorithms with a good node-averaged complexity in Section 5.

We now define a total order on the layers of a (γ, ℓ, L)-decomposition in the natural way. This will be useful in the design of our algorithm in Section 5.

Definition 8 (layer ordering)**.** We define the following total order on the (sub)layers of a (γ, ℓ, L)-decomposition.

- $V^R_{i,j} < V^R_{i',j'}$ iff $i < i' \vee (i = i' \wedge j < j')$
- $V^R_{i,j} < V^C_i$
- $V^C_i < V^R_{i+1,j}$

Accordingly, we will use terms such as "lower layer" to refer to a layer that appears earlier in the total order than some considered other layer.

For the interested reader, in Section 7.1, we provide some more intuition about the function $f_{\Pi,k}$, and about how the existence of this function is related with the complexity of a problem.

5 Algorithm for Intermediate Worst-Case Complexity Problems

In this section, we provide an algorithm with node-averaged complexity $O(\log^* n)$ on bounded-degree trees for all LCLs that can be solved in worst-case complexity $O(\log n)$ on bounded-degree trees. The high-level idea of our algorithm is similar to the strategy in the generic algorithm of Section 4.2: compute a (γ, ℓ, L)-decomposition, propagate types up through the layers, and then propagate a choice of labels through the layers in reverse order. Importantly, in order to obtain a good node-averaged complexity, we will not wait for the decomposition to be completely computed, but instead we show that we can allow a sufficient amount of nodes to terminate early. A crucial ingredient for achieving this is identifying some nodes with nice properties which we promote to be a *local maximum* (that is, nodes that have a layer number that is strictly higher than all their neighbors), allowing it and many other nodes to terminate.

However this approach requires us to find a suitable (γ, ℓ, L)-decomposition that in particular guarantees that sufficiently many nodes become local maxima. Observe that the standard decomposition algorithm may not guarantee this property: in a balanced binary tree for example, only the root would become a local maximum. To achieve the required property we will compute an altered decomposition that in some sense leaves a bit of extra space between two compress layers. We will use this extra space to insert new compress paths creating local maxima. Here we need to take special care that such a promotion is possible while guaranteeing that at the end we still have a valid (γ, ℓ, L)-decomposition.

As this part is quite long and it is possible to get lost in the details, we next provide an overview of the structure of this section.

- **Section 5.1:** The Decomposition Algorithm
  - Stating a centralized version of the algorithm. Definitions 9 to 11
  - Proving that it computes a valid (γ, ℓ, L)-decomposition and thereby proving correctness. Definitions 12 and 13 and Lemma 14

  - Proving that we need $O(\log n)$ iterations. Lemmas 15 and 16 and Corollary 17

- **Section 5.2:** Local Maxima and Bounding Quality
  - Decomposing the tree into subtrees. Definitions 18 and 19 and Lemma 21
  - The promotion creates local maxima and they account for a large amount of quality. Lemmas 22 to 26
  - Upper bounding the quality of a free node. Lemmas 27 to 29.

- **Section 5.3:** Distributed Algorithm and Node Averaged Complexity
  - The distributed implementation and the round complexity of an iteration. Lemma 30
  - Proving $O(\log^* n)$ node-averaged complexity. Lemmas 31, 33 and 34 and Corollary 32

5.1 The Decomposition Algorithm

Our algorithm still does rake and compress each iteration and we will assign layers for a decomposition along the way. Clearly during the execution of the algorithm we will not yet have a complete (γ, ℓ, L)-decomposition. Say we only have layers up until the i-th layer, $V_1^R, \ldots, V_i^R$ and $V_1^C, \ldots, V_i^C$, that already satisfy the definition of a (γ, ℓ, L)-decomposition. The definitions we provide already apply to graphs with a partial decomposition, i.e., a partial assignment of nodes to layers of the form $V_{i,j}^R$ and V_i^C (for some positive integers i, j). A bit later we give a formal definition of a partial decomposition. For now we start by distinguishing between nodes that have already been assigned a layer and those that have not.

Definition 9 (free and assigned nodes)**.** We call a node that has not been assigned to a layer a *free* node. A node that has been assigned to a layer is called *assigned*.

Let G denote our input tree and assume that a subset of nodes has already been assigned to some layers. Let G' denote the subgraph of G induced by all assigned nodes. View Figure 3 for the general picture that should be kept in mind for the rest of this section.
We now define two notions that will be crucial for our approach. Recall that we have a total ordering on the layers of a decomposition due to Definition 8 (that naturally extends to partial decompositions).

Definition 10 (local maximum)**.** A *local maximum* is an assigned node $v \in V(G')$ with the following two properties:

1. Node v and all of its neighbors are assigned, i.e., they are all contained in $V(G')$.
2. For each neighbor w of v, the layer of w is strictly smaller than the layer of v.

In our algorithm, we will artificially promote some nodes to a higher layer in order to produce local maxima. The second notion allows to measure how efficient a node would be as a local maximum. The quality of a node v is the number of nodes that are depending only on v to pick its label so they can pick their labels. So if v terminates and chooses an output all of these nodes can also terminate.

To keep track of this, we will additionally assume that (some of the) edges of the input graph can be oriented see Figure 4. Then our algorithm will orient the edges in such a manner that any node u will have an oriented path from v to u if and only if u will be able to terminate if v does. In practice this means if u is raked away we orient the single edge from u's parent towards itself

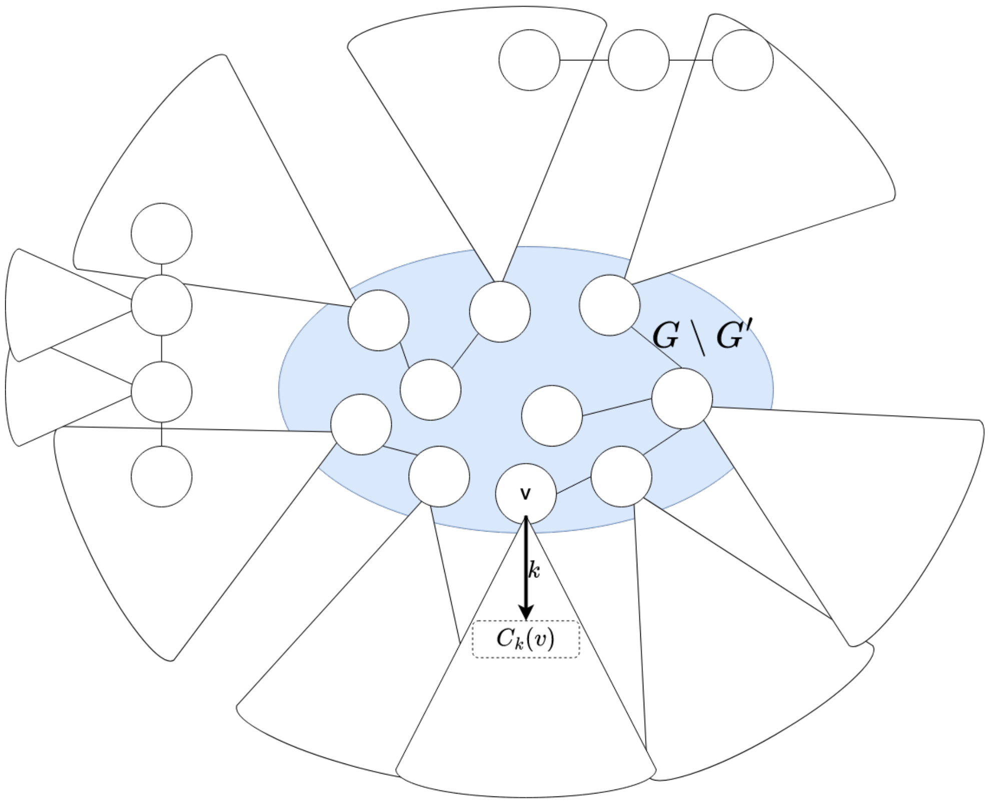

Figure 3: A picture during the execution of a rake and compress like algorithm. We have in blue the graph of remaining free nodes. All of the already assigned nodes are hanging in subtrees from the nodes of $G \setminus G'$. The compress paths are not part of any subtrees and connect the components of free nodes. Also the set $C_k(v)$ is exactly the nodes at distance k from v that are in the tree hanging from v.

and orienting the ends of compress paths inwards. If an edge is not explicitly oriented it is not considered oriented at all. For some given v this orientation now defines $H(v)$ a subgraph of nodes that can be reached over oriented paths. This graph is then exactly all of the nodes that are only waiting for v to choose an output and hence these could all terminate if v became a local maximum.

Definition 11 (quality)**.** For any node $v \in V(G)$, let $H(v)$ denote the set of all nodes w that can be reached from v via a path $(v = v_0, v_1, \dots, v_j = w)$ such that the following hold:

1. The edge $\{v_{i-1}, v_i\}$ is oriented from v_{i-1} to v_i, for each $1 \le i \le j$.

2. All nodes on the subpath from v_1 to w are assigned, i.e., they are all contained in $V(G')$.

3. The layer of v_i is smaller than or equal to the layer of v_{i-1}, for each $2 \le i \le j$, and if v_0 is assigned, then the layer of v_1 is smaller than or equal to the layer of v_0.

If v is a local maximum, or a descendant of a local maximum, then the quality $q(v) := 0$. Otherwise the *quality* $q(v)$ of a node v is the number of nodes in $H(v)$, i.e., $q(v) := |H(v)|$.

We chose the name quality, since we will later decide on which nodes to fix by trying to maximize this quantity. We will later also see that the sum over the quality of all free nodes are exactly all of the nodes that are yet to terminate. So by giving an upper bound on $\sum_{v \in G \setminus G'} q(v)$ we will be able to argue about the progress of our algorithm.

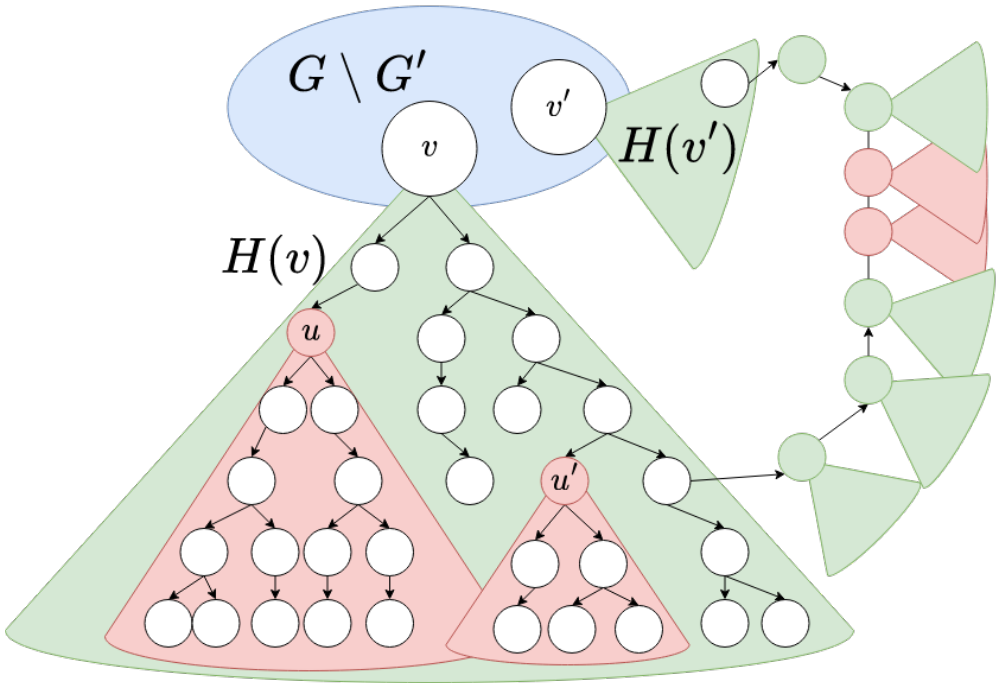

Figure 4: Illustrating the quality of a node. v and v' are free nodes inside of $G \setminus G'$. $H(v)$ (respectively $H(v')$) are all nodes inside the green cone attached to v (v') and the green nodes in the compress path. u and u' are local maxima, so because of point 3 in Definition 11 they and the red trees hanging from them do not contribute to $H(v)$ (respectively $H(v')$.

The algorithm. We give two versions of the algorithm first. Algorithm 1 provides an informal writeup of the highlevel idea and then Algorithm 2 gives all the details. In previous work the rake and compress procedure is done by performing rakes first and then compresses after the rakes are done. We need to change the ordering, but to still get the same guarantees, we start with a round of γ rakes. Then in each iteration we first perform a modified compress operation. Instead of taking paths that are as short as possible, we make sure we have some extra slack at the end of each compress path. This is to ensure that compress paths are always far enough away from nodes that we want to promote. We then make sure that all of these slack nodes are raked away nicely, by performing a set of γ rakes right after our modified compress procedure.

Now we are ready for the part where we actually promote some nodes to be local maxima, based on the following definition. For each node $v \in V(G)$, let $C_k(v)$ denote the set of assigned nodes w that have distance exactly k from v and for which all internal nodes of the unique path from v to w are assigned as well, i.e.,

$$C_k(v) = \{w \in V(G') \mid \text{dist}(v, w) = k,\ x \in V(G') \text{ for all } x \neq v \text{ on the path from } v \text{ to } w\}.$$

Refer to Figure 3 for an example. We determine the node that we want to promote by choosing the node v^* that has the highest quality among nodes in $C_b(r)$. This ensures that a lot of nodes can terminate early. We are however not always able to promote v^*, but we will later see, that when we are not able to, then this is because we have very recently been successful in promoting another close-by node. One important thing to note is that our algorithm proceeds with only even iteration numbers. This is because in a single iteration we modify layers with three different indexes. For each iteration i (where i is an even number), we further define sets $G^{(i)}$ and $G^{(i+1)}$ to differentiate between two points during the execution of iteration i of the algorithm. The details of the algorithm

Algorithm 1: Compute Decomposition *Informal*

```
Input: $G = (V, E), \Pi$
1  compute $\ell$ from $\Pi$
2  $b \leftarrow \ell + 2$
3  $\gamma \leftarrow \ell + 3$
4  Perform $\gamma$ orienting Rakes
5  for $O(\log n)$ times (until every node is assigned to a layer) do
6      for each long enough path do
7          Ignore the first and last $\gamma$ nodes                ▷ Will be raked away.
8          Perform normal compress on the truncated path
9          Orient the ends inwards
10     Perform $\gamma$ orienting Rakes        ▷ Thereby raking away the slack from compress
11     for each free node $r$ do
12         $v^*$ is the child at distance $b$ with the largest quality
13         if The path from $r$ to $v^*$ does not intersect with any compress path then
14             Promote $v^*$ into a local maximum by turning the path from $r$ to $v^*$ into a
                 compress path
```

are given as pseudocode in Algorithm 2. Note that Algorithm 2 provides a description of the steps of the algorithm without specifying how the algorithm is implemented in a distributed manner. We will take care of the latter in Section 5.3.

During the analysis of the algorithm, we often talk about different parts of the algorithm and reference certain times during the execution. To make this a bit more formal, we introduce *static points.*

Definition 12. We call a point in time during the execution of Algorithm 2 that is not inside an inner for-loop[5] a *static point* (in time).

We can use this definition to now prove that at any static point t our algorithm has a valid partial decomposition. For this we extend the definition of a (γ, ℓ, L)-decomposition to graphs where only a subset of the nodes have been assigned to layers. It is almost an exact restatement of Definition 6, with the subtle difference being that we only care about a subset $V' \subseteq V(G)$ of nodes. As a reminder we state the entire definition again.

Definition 13 (partial (γ, ℓ, L)-decomposition)**.** A *partial* (γ, ℓ, L)*-decomposition* of a graph G is a partition of some subset $V' \subseteq V(G)$ into $2L - 1$ layers $V_1^R = (V_{1,1}^R, \ldots, V_{1,\gamma}^R), \ldots, V_L^R = (V_{L,1}^R, \ldots, V_{L,\gamma}^R)$, $V_1^C, \ldots, V_{L-1}^C$ such that the following hold.

1. Compress layers: The connected components of each $G[V_i^C]$ are paths of length in $[\ell, 2\ell]$, the endpoints have exactly one neighbor that is in a higher layer or free, and all other nodes do not have any neighbor that is in a higher layer or free.

2. Rake layers: The diameter of the connected components in $G[V_i^R]$ is $O(\gamma)$, and for each connected component at most one node has a neighbor that is in a higher layer or free.

3. The connected components of each sublayer $G[V_{i,j}^R]$ consist of isolated nodes. Each node in a sublayer $V_{i,j}^R$ has at most one neighbor in a higher layer or sublayer.

[5]Inner for-loops refers to the for loops in Lines 12, 22, and 26.

Algorithm 2: Compute Decomposition

```
   Input: $G = (V, E), \Pi$
1  compute $\ell$ from $\Pi$
2  $b \leftarrow \ell + 2$
3  $\gamma \leftarrow \ell + 3$
4  $G^{(0)} \leftarrow G$
5  $N^{(0)} = \emptyset$
6  for $j = 1$ to $\gamma$ do
7      add every node of degree 0 or 1 in the graph induced by the free nodes to $V^R_{1,j}$
8      if the degree was 1, have each such node orient its unique incident edge in that graph
         towards itself
9  $G^{(1)} \leftarrow G \setminus V^R_1$
10 $i \leftarrow 2$
11 for $O(\log n)$ times (until every node is assigned to a layer) do
12     for each maximal path $P$ consisting of nodes of degree exactly 2 in the graph induced by
         the free nodes do
13         if $|V(P)| \geq 4\ell + 9$ then
14             $P' \leftarrow$ the subpath of $P$ consisting of all nodes that have distance at least $\ell + 3$
                 from both endpoints of $P$
15             compute a subset $Z \subseteq V(P')$ such that no two nodes in $Z$ are adjacent, no
                 endpoint of $P'$ is in $Z$, and every maximal subpath of $P$ consisting only of nodes
                 in $V(P') \setminus Z$ has length in $[\ell, 2\ell]$
16             add every node in $V(P') \setminus Z$ to $V^C_{i-1}$
17             add every node in $Z$ to $V^R_{i,1}$
18             let $P' = (v_0, \ldots, v_w, \ldots, v_y, \ldots v_j)$, and let $v_w \in Z$ be the first node on $P'$ that is
                 in $Z$ and $v_y \in Z$ the last.
19             orient the edges in the path $(v_0, \ldots, v_{w-1})$ from $v_0$ towards $v_{w-1}$
20             orient the edges in the path $(v_{y+1}, \ldots, v_s)$ from $v_s$ towards $v_{y+1}$
21             $N^{(i)} \leftarrow N^{(i-2)} \cup \{v_w, v_{w+1}, \ldots, v_{y-1}, v_y\}$          ▷ add all nodes that have no edge
                 oriented towards them
22     for $j = 1$ to $\gamma$ do
23         add every node of degree 1 in the graph induced by the free nodes to $V^R_{i,j}$
24         have each such node orient its unique incident edge in that graph towards itself
25     $G^{(i)} \leftarrow G \setminus \left(\bigcup_{1 \leq a \leq i} V^R_a \cup \bigcup_{1 \leq a \leq i-1} V^C_a\right)$          ▷ all free nodes
26     for each node $r \in G^{(i)}$ do
27         $v^* \leftarrow \arg\max_{v \in C_b(r)} q(v)$          ▷ breaking ties arbitrarily
28         $P'' \leftarrow$ the path from $v^*$ to $r$
29         if no node in $V(P'')$ is currently assigned to a layer of the form $V^C_a$ for some
             $1 \leq a \leq i - 1$ then
30             remove every node in $V(P'') \setminus \{v^*, r\}$ from the layer it is currently assigned to
                 and add it to $V^C_i$
31             remove $v^*$ from the layer it is currently assigned to and add it to $V^R_{i+1,1}$
32             mark $v^*$ as promoted
33     $G^{(i+1)} \leftarrow G \setminus \left(\bigcup_{1 \leq a \leq i+1} V^R_a \cup \bigcup_{1 \leq a \leq i} V^C_a\right)$          ▷ all free nodes
34     $i \leftarrow i + 2$
```

We note that at all times V_i^R is to be understood as the union of all nodes in the layers $V_{i,j}^R$, i.e.,

$$V_i^R := \bigcup_{1 \leq j \leq \gamma} V_{i,j}^R.$$

We now prove that during our algorithm we really have a valid partial decomposition.

Lemma 14. *Consider any static point t in time during the execution of Algorithm 2, and let i be the iteration in which t occurs, where we set i to 2 if t occurs before the first execution of Line 11. Then the partial assignment of nodes to layers at time t forms a partial $(\gamma, \ell, i+1)$-decomposition.*

Proof. We prove the lemma by induction in all static points t in time. For the base case observe that the statement trivially hold at the start of the algorithm (where all layers are empty sets). For the induction step, assume that the lemma statement holds at some static point t in time. We will show that it then also holds at the next static point in time, which we will call t'.

If the changes to the decomposition between time t and t' are given by Line 7, by Lines 16 and 17, or by Line 23, then the lemma statement at time t' follows in a straightforward way from the design of the algorithm and the induction hypothesis.

Hence, consider the (only other) case that the change to the assignment of the nodes to layers between times t and t' is due to Lines 30 and 31. From the design of Lines 30 and 31 and the fact that, before the execution of Line 30, nodes have only been added to lower layers than V_i^C, it follows that all nodes who are added to V_i^C during the execution of Line 30 satisfy Property 1 in Definition 13 at time t'. Similarly, node v^* added to layer $V_{i+1,1}^R$ (or rather its connected components in V_{i+1}^R, resp. $V_{i+1,1}^R$) satisfies Properties 2 and 3 of Definition 13 at time t', as it has only neighbors of lower layers. It remains to show that the properties of Definition 13 also (continue to) hold at time t' for all layers that are lower than V_i^C.

To this end, consider the path P'' from v^* to r (that the behavior of Algorithm 2 between time t and t' depends on). By the induction hypothesis, each node in $V(P'') \setminus \{r\}$ has at most one neighbor at time t that is in a higher layer or free, and no node assigned to a layer of the form V_a^R for some a has a neighbor in the same layer. Since r is free (while all other nodes in P'' are not free, due to the definition of $C_b(r)$, which Line 27 depends on) and no node in $V(P'') \setminus \{r\}$ is assigned to a layer of the form V_a^C (due to the condition in Line 29), it follows that the layers of the nodes increase strictly from v^* to the last node of P'' before r. This implies that, at time t, each node that is not free but has a neighbor in $P'' \setminus \{r\}$ while not being contained in P'' itself is assigned to a strictly lower layer than the aforementioned neighbor, due to Definition 13 and the induction hypothesis. Since this fact also holds at time t' (as each node in P'' is assigned to a higher layer at time t' than at time t) and P'' does not contain any node assigned to a layer of the form V_a^C at time t (due to the condition in Line 29), it follows that the properties of Definition 13 continue to hold at time t' for all layers lower than V_i^C (where we use that they hold at time t due to the induction hypothesis). This concludes the induction.

As the highest layer at time t is V_{i+1}^R, the lemma statement follows. □

Next we argue that our algorithm behaves in a similar way to a normal rake and compress algorithm. We use the following lemma.

Lemma 15 ([16])**.** *Given a tree with n nodes, by performing α rakes*[6] *and 1 compress*[7] *with minimum path length β, the number of remaining nodes is at most $\frac{\beta}{2\alpha} n$.*

[6]A rake operation is the removal of all nodes of degree 0 or 1.

[7]A compress operation consists of the removal of all paths of nodes of degree exactly 2, that are of length at least β.

Now to show that our algorithm fulfills the conditions for this lemma every few iterations.

Lemma 16. *There exists some constant $0 < \varepsilon < 1$ such that, for any even positive integer $i \geq 10$, the number of free nodes after iteration i of Algorithm 2 is at most $n\varepsilon^i$.*

Proof. In our algorithm, the minimum path length is $\beta = 4\ell + 9$, but not all of these nodes are immediately put into a compress layer. Instead we leave γ nodes at both ends. However all of these nodes that were left are immediately raked away in Line 22. So for every even i all nodes in paths of length at least β are assigned a layer and therefore a compress is performed. Moreover, for every even i, γ rakes are performed. This implies that for every iteration $10i + j$ ($j \in \{0, 2, 4, 6\}$), we perform γ rakes, for a total of $4\gamma = 4\ell + 12$ rakes, and then, in iteration $10i + 8$, we perform a compress operation. Hence, by Lemma 15, the number of nodes that we have at iteration $10(i + 1)$ are at most $\frac{b}{8\gamma} \leq 1/2$ times the number of nodes that we have at iteration $10i$. Hence, at iteration i, for i integer multiple of 10, we get that the number of remaining nodes is at most $n/2^i$. In order to obtain a smooth progression, we pick $\varepsilon = \frac{1}{\sqrt[20]{2}}$. Observe that, for all $i \geq 10$, $n\epsilon^i \geq n/2^i$, and hence the claim follows. □

As a result of this Lemma 16 we immediately get that we will be done in at most $O(\log n)$ rounds and because of Lemma 14 we also get that we will have a complete (γ, ℓ, L)-decomposition at the end.

Corollary 17. *The decomposition (i.e., the assignment of nodes to layers) produced by Algorithm 2 is a (γ, ℓ, L)-decomposition, where γ and ℓ are the parameters from Algorithm 2 and L is a suitably chosen value from $O(\log n)$.*

In Section 5.3 we will then show that local maxima allow us to have nodes terminate early, such that we will obtain a good node-averaged complexity at the end. In the next section our main goal is to prove that enough of these local maxima actually exist and are nicely distributed in the graph.

5.2 Local Maxima and Bounding Quality

We will first see that during the execution of our algorithm we can actually decompose the graph into two parts. First the free nodes and all already raked nodes hanging from them and second the nodes that are in compress paths who also have raked nodes hanging from them see Figure 3 from before. In compress paths there already are *natural* local maxima just from the way the original decomposition algorithm works. These local maxima are sufficient to have all of the compress path terminate, except for the very ends. These ends are oriented such that they are charged to the quality of some remaining free node. We will see that by summing over the quality of all free nodes we get exactly all of these nodes that are not yet descendant of a local maximum.

We start with the following definitions with regards to the orientation of the nodes. Intuitively nodes are oriented, such that if a node is raked away, the edge is oriented from a parent towards the just removed node and the ends of compress paths are oriented inwards. This is illustrated in Figure 5. Let i be some arbitrary even positive integer, and consider the input tree G together with the partial assignment of nodes to layers obtained after Line 25 and the corresponding set $G^{(i)}$ in iteration i. Again, let G' denote the subgraph of G consisting of all nodes that have already been assigned a layer. We can express $G' = G \backslash G^{(i)}$.

Definition 18 (child, parent, descendant, ancestor, orphan)**.** For any edge $\{w, w'\}$ oriented from w to w', we call w' a *child* of w and w the parent of w'. For any oriented path $(w, \ldots, w')$ that is consistently oriented from w to w', we call w' a *descendant* of w and w an *ancestor* of w'. We call a node with no edges oriented towards itself an *orphan*.

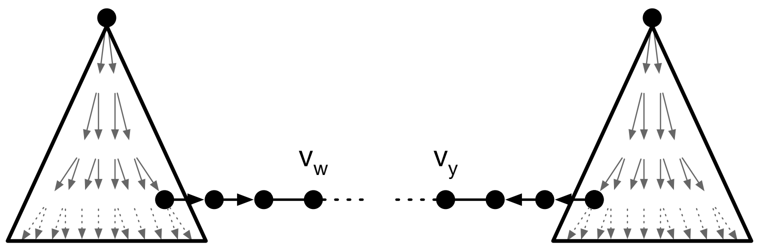

Figure 5: An example of how edges are oriented. The two nodes at the top are free nodes and they are connected by a compress path. The nodes inside of the respective trees are oriented from the root towards the leaves. The ends of the compress path are oriented inwards, however the middle part of the compress path is not oriented in any way, as illustrated at the innermost nodes, the incident edges of which are no longer oriented.

We give one of the most important definitions, that of a subtree of assigned nodes. We illustrate how they exist in relation to $G^{(i)}$ in Figure 6.

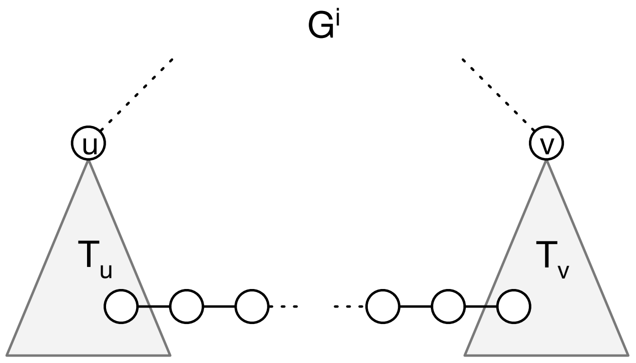

Figure 6: The graph induced by $G^{(i)}$ is not necessarily connected, it might happen that some two nodes $u \in G^{(i)}$ and $v \in G^{(i)}$ are both having a subtree of assigned nodes hanging from them. In this illustration we name them T_u and T_v respectively. These trees are then connected by a compress path.

Definition 19 (subtree of assigned nodes)**.** For any node $v \in V(G)$ and any positive integer i, the subtree of assigned nodes $T^{(i)}(v)$ denotes the set that contains v and all nodes w that can be reached from v via a path $(v = v_0, v_1, \dots, v_j = w)$ with the following property: at the time during the execution of Algorithm 2 when $G^{(i)}$ is defined (i.e., Line 25 for even i, Line 9 for $i = 1$, and Line 33 for larger odd i), the edge $\{v_{a-1}, v_a\}$ is oriented from v_{a-1} to v_a, for each $1 \leq a \leq j$.

We denote by $h^{(i)}(v)$ the height of $T^{(i)}(v)$, formally

$$h^{(i)}(v) = \max\{\text{dist}(v, u)\}_{u \in T^{(i)}(v)}.$$

To make the definition of $T^{(i)}(v)$ a bit clearer, we are going to make some small observations about these trees.

Observation 20. The following hold:

1. **Orientations** of the edges are only set once and then will not be changed anymore. This holds true simply because edges are only oriented when a node is first assigned a layer and not when layers are changed later. Additionally each node only ever orients one edge toward itself. For an illustration of the orientation refer to Figure 5

2. **Compress Paths**: For a path P' that is handled in the loop of Line 12, only the first and last few nodes of the path are oriented. As a result the nodes between the first and last nodes v_w and v_y only have incident edges that are oriented away from them (and unoriented incident edges). Therefore all of the nodes on the unique path $v_w, \ldots, v_y$ are orphans.

3. **Promoted Compress Paths**: For a path that has its layers changed inside of the if statement in Line 29, the orientation stays unchanged.

4. **The Set** $N^{(i)}$ defined in Line 21 only contains nodes that have no edge oriented towards them.

We now show how we can express the entire input graph G in terms of these trees. For this we will make use of the sets $N^{(i)}$ which contain all of the nodes that were assigned during the compress procedure. Intuitively any node that is already assigned a label has to hang in some subtree of assigned nodes. By taking a union over all of the subtrees hanging from $G^{(i)}$ and $N^{(i)}$ we will cover the entire tree. Also note that in Algorithm 2, $N^{(i)}$ is technically only defined for even values of i. We additionally define $N^{(i+1)} = N^{(i)}$ (for even i) such that we do not have to worry about only referring to the correct set.

Lemma 21. *For each positive integer i, the following holds:*

$$V(G) = \left(\bigcup_{v \in G^{(i)}} T^{(i)}(v) \right) \cup \left(\bigcup_{v \in N^{(i)}} T^{(i)}(v) \right)$$

Furthermore the two big unions are disjoint.

Proof. We call a node v an *orphan* if it has no edge oriented towards it. Now any node that has an edge oriented towards it, is clearly hanging in a tree rooted at an orphan node. So as long as we have all trees of nodes that are orphans, we cover the entirety of $V(G)$. The only nodes that are orphans are the nodes of $G^{(i)}$ and the nodes in $N^{(i)}$, since all other nodes had an edge oriented towards them, when they were assigned to a layer.
Since v is in its own subtree of assigned nodes we trivially get that all nodes of $G^{(i)}$ are in $\bigcup_{v \in G^{(i)}} T^{(i)}(v)$. And the same way we get that also all nodes of N are in $\bigcup_{v \in N^{(i)}} T^{(i)}(v)$.
We see that the two are disjoint, because $G^{(i)}$ and $N^{(i)}$ are disjoint. Furthermore any node has at most one edge oriented towards itself and it therefore also only belongs to one subtree of assigned nodes, rooted at an orphan. □

Notice that for each of these subtrees of assigned nodes, the quality of the root counts exactly how many nodes in this tree still need to terminate. So by giving a good upper bound on the quality of nodes in $G^{(i)}$ we will be able to show that in each iteration a constant fraction of the remaining nodes terminate.

Creating Local maxima For the next lemmas, we narrow down our view to some concrete iteration i and will hence drop some of the (i) in the exponents. We prove that if the statement in Line 29 is true and we promote v^* to layer $V^R_{i+1,1}$ we get that v^* will indeed be a local maximum.

Lemma 22. *After v^* is put into $V^R_{i+1,1}$ by Line 31, v^* will be a local maximum. Additionally for any node u that is a local maximum, u will always stay a local maximum and none of the nodes in its assigned subtree $T^{(i)}(u)$ will change their layer during the rest of the algorithm.*

Proof. First, we show that v^* actually becomes a local maximum when it is put into $V^R_{i+1,1}$. Since v^* was already in some layer beforehand, clearly all of its children must also have been assigned nodes at that point in time. Furthermore, v^* is assigned to layer $V^R_{i+1,1}$ which no other node is assigned to yet and is clearly the highest layer up until this point. So v^* will be in a higher layer than all of it's children. That just leaves the parent p of v^*, but p must be an assigned node on the path to the root node r and therefore is assigned layer V^C_i, which implies that v^* becomes a local maximum.
Now we prove that any node u that is a local maximum, will remain a local maximum. The only way that it can happen that u stops being a local maximum, is if one of its neighbors gets assigned a new layer. The only lines in the algorithm that assign new layers to already assigned nodes are Lines 30 and 31. But the quality of $q^{(i)}(u) = 0$, due to it being a local maximum and also the quality of all of the nodes in $T^{(i)}(u)$ will be 0. So they will never be chosen as a node to be promoted. So now the only possible case is that the parent node p of u gets promoted. For this we make a case distinction between the different ways u might have become a local maximum. There are three ways how this could have happened.

1. u was promoted in Line 31, then its parent p was changed into a compress layer. Therefore if there was an attempt to promote p, the condition in Line 29 would not hold. So p cannot be promoted.

2. u became a local maximum by being part of Z during the compress step, then u has no parent p

3. u was a node of degree 0 before it was removed, but then also u does not have a parent.

□

When we encounter such local maxima during the analysis of our algorithm we will have to make the distinction whether or not they became local maxima in Line 31 or Line 17. The next lemma will help us make that distinction.

Lemma 23. *When a node v becomes a local maximum in Line 17 of iteration i, then it has distance at least $\gamma + 1$ from any node in $G^{(i)}$ as defined in Line 25. Additionally, all nodes that are assigned to V^C_{i-1} in Line 16 have distance at least $\gamma + 1$ from any node in $G^{(i)}$.*

Proof. For v to be assigned to $V^R_{i,1}$ it has to be in P'. Any node in P' has distance at least $\ell + 3$ from the endpoints of P. After all nodes of P' have been assigned either to layer $V^R_{i,1}$ or to layer V^C_{i-1}, $P \setminus P'$ will consist of two disjoint paths of length $\ell + 2$ each. Now both of these paths are going to be raked away completely in the $\gamma = \ell + 3$ rakes performed in line 22. As a result any node that is in P' will have at least such a path of $\ell + 3 = \gamma$ raked nodes between itself and any node that is free after Line 25. As those latter nodes precisely form $G^{(i)}$, the first statement in the lemma follows. The second statement follows by the same argument. □

Additionally we will need this small lemma to argue about the orientation of the paths to our promoted nodes.

Lemma 24. *For all iterations i, if there exists a Path P of nodes that have a layer assigned from a free node r to a node v that was already assigned a layer and this path has length at most b then v is a descendant of r.*
Additionally if a node v^* is promoted in line 31, both it and the entire path P'' are descendants of r.

Proof. Remember that an orphan is a node that has no edge oriented towards it and only free nodes and nodes in $N^{(i)}$ are orphans. Let v be a node that was already assigned a layer and let P be a path from r to v of length at most b. We need to show, that all edges are oriented from r to v. By Lemma 23 we get that none of these nodes were assigned a layer inside the compress loop in line 12. As a result all of them must have been assigned a layer at line 8, or line 24. An edge oriented in any of these lines is always oriented towards a free node and therefore towards an orphan. As a result all of the edges in P must have some orientation.
Clearly every node that is not an orphan has a path that is oriented from an orphan to itself, since orientations are only ever made towards orphans. Then since r is an orphan, $\{r, v_0\}$ must therefore be oriented towards v_0.
Now assume for contradiction that the edges inside P are not oriented from r to v, then there must exist a first edge $\{v_j, v_{j+1}\}$ that is oriented in the other direction, so from v_{j+1} to v_j. But now v_{j+1} would have to edges oriented away from it, which cannot happen. The fact that the unique path $r, v_0, \ldots, v_{b-1}, v^*$ is oriented from r to v^* follows trivially. □

Notice that there is a case distinction in Line 29; we use the above two lemmas to prove that if the statement in Line 29 is true and we promote v^* to layer $V^R_{i+1,1}$ we get that $q(v^*)$ will be a large part of $q(r)$. Thereby showing that with each promotion we reduce the quality by a constant fraction. Recall the definitions of the quality $q(v)$ of a node v and $H(v)$, provided in Definition 11.

Lemma 25. *If the condition in Line 29 holds and v^* is promoted, then $q(v^*) \geq \frac{q(r)}{2\Delta^b}$ when Line 29 is evaluated.*

Proof. The statement follows from the fact that

$$q(r) \leq \Delta^b + \sum_{v \in C_b(r)} q(v)$$

Now this is true, because we can separate $H(r)$ into the nodes that are close and those that are far. Concretely a node u is close, if the path to r is strictly less than b. In this case we get by Lemma 24 that all of these close nodes are actually descendants of r. But only Δ^b such nodes can exist. A node u is far, if the path to r is at least b nodes long, at which point it has to pass through one of the nodes $v \in C_b(r)$. Now since $u \in H(r)$ the unique path from r to u satisfies the criteria for u to be in $H(r)$, then the subpath from w to u must also satisify these criteria and u is therefore included in $q(w)$. We then get

$$q(r) \leq \Delta^b + \sum_{v \in C_b(r)} q(v) \leq \Delta^b + |C_b(r)| \cdot q(v^*) \leq \Delta^b + \Delta^b q(v^*) \leq 2\Delta^b q(v^*).$$

As a result $u \in H(v)$ and therefore u is accounted for in $q(v)$. The second inequality comes from the fact v^* has the highest quality among nodes in $C_b(r)$. □

However the condition in Line 29 may not hold in every iteration. So if it does not hold, we show that there must exist some node x promoted (earlier) in Line 32 that is close to v^*.

Lemma 26. *If the condition in Line 29 does not hold, then there exists a promoted node x at distance at most $2b-1$ from r, such that $q(x) \geq \frac{q(v^*)}{2\Delta^b}$ immediately before x was promoted.*

Proof. As the intuition of this proof is a lot easier to grasp visually we have provided a sketch of this scenario in Figure 7. Consider the path P'' from r to v^*. Since the condition in Line 29 is not true, some node $u \in V(P'') \setminus \{r\}$ must already be in a compress layer (i.e., a layer of the form V^C_a),

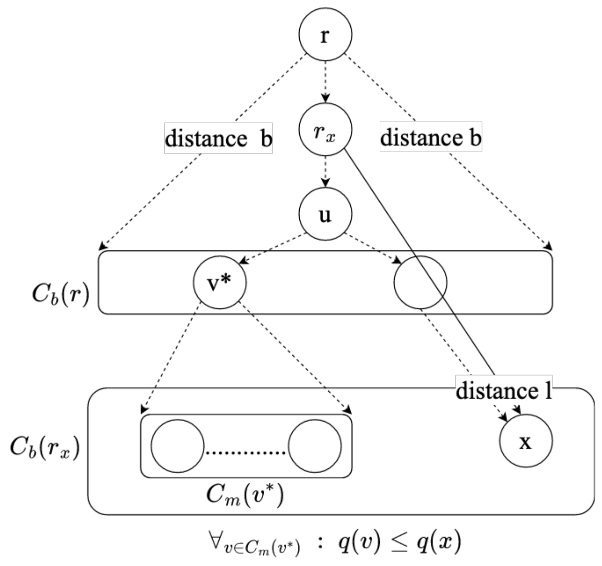

Figure 7: v^* cannot be fixed, because x is already fixed. u is the common ancestor and r_x was root, when x was fixed. Clearly the way m is chosen, $C_m(v^*)$ is a subset of $C_b(u)$

with $\text{dist}(u, v^*) < b$. By Lemma 23 and the fact that $\gamma = b + 1$, we have that this compress node u must have been added as the result of Line 30, hence there must also be a node x that was actually promoted in Line 31. Now because of Lemma 24 u is an ancestor of both v^* and x. Then we have that $\text{dist}(u, x) \leq b$ and $\text{dist}(u, v^*) < b$. Now let r_x be the node that was the free node which decided to promote x, so $\text{dist}(r_x, x) = b$, then $\text{dist}(r_x, v^*) < b$. Now define $m = d(r_x, x) - d(r_x, v^*) < b$, then we get that $C_m(v^*) \subset C_b(r_x)$ and since x was the choice of r_x, we get that

$$\forall_{v \in C_m(v^*)}\, q(v) \leq q(x)$$

Now to bound $q(v^*)$

$$q(v^*) \leq \Delta^m + \sum_{v \in C_m(v^*)} q(v) \leq \Delta^b + \Delta^b q(x) \leq 2\Delta^b q(x)$$

□

Upperbounding the quality of a free node

The next lemma will be the main technical result that shows that enough nodes are in subtrees of local maxima. We will show this implicitly by upperbounding the quality of the remaining free nodes. However we will have to introduce some more notation. We are partitioning each assigned subtree $T^{(i)}(v)$ into $\alpha = \left\lceil \frac{h^{(i)}(v)+1}{b} \right\rceil$ subsets $S_0^{(i)}(v), \ldots, S_{\alpha-1}^{(i)}(v)$, where, for each $0 \leq j \leq \alpha - 1$,

$$S_j^{(i)}(v) := \{u \in T^{(i)}(v) \mid b \cdot j \leq \text{dist}(u, v) < b \cdot (j + 1)\}.$$

So every b layers of the tree are grouped into one such set. This is illustrated in Figure 8.

Clearly,

$$\bigcup_{0 \leq j \leq \alpha - 1} S_j^{(i)}(v) = T^{(i)}(v).$$

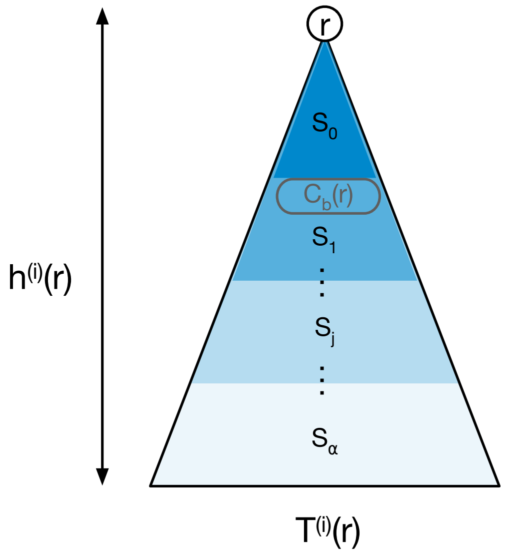

Figure 8: A tree rooted at r is split into the different layer sets $S_0, \ldots, S_\alpha$. Since r in included in S_0 the union of those layer sets will form the entire tree.

One useful equality following from these definitions is

$$\bigcup_{v \in C_b(r)} S_j^{(i)}(v) = S_{j+1}^{(i)}(r),$$

which holds for any assigned node r and any $1 \leq j \leq \alpha - 1 = \left\lceil \frac{h^{(i)}(r)+1}{b} \right\rceil$.

For convenience, we also formally define $S_j^{(i)}(v)$ for indices that are larger than $\alpha - 1$: we set $S_j^{(i)}(v) := \emptyset$ for all $j \geq \alpha$.

Now we are ready to give the main result about the quality of free nodes. Essentially we want to look at assigned subtrees that are hanging from free nodes. The main intuition is that the deeper layers contain more nodes and also contain more local maximums. This is simply as a result of them being in the removed part for more iterations. So if we were able to proof that for each layer j it was true, that only $\lambda^j |S_j^{(i)}|$ nodes remain in that layer for some constant fraction λ, then this would be enough for constant node averaged complexity. However such a statement is simply not true. This is because nodes are chosen based on their quality and this might result in some layers that have almost no nodes fixed. But then this implies that these layers are very sparse. What we will instead be able to show is that in each subtree of assigned nodes the number of remaining nodes (or equivalently the quality of the root) satisfy an inequality that expresses almost the same thing.

Lemma 27. *There exists a constant $0 < \lambda < 1$ (that only depends on Π and Δ) such that for all even positive integers i, the following inequality holds at the end of iteration i, for all nodes $r \in G^{(i+1)}$:*

$$q^{(i+1)}(r) \leq \sum_{j=0}^{\lceil (h^{(i+1)}(r)+1)/b \rceil - 1} \lambda^j |S_j^{i+1}(r)|$$

One big problem in the proof of Lemma 27 is that the statement holds for free nodes only and as we will see in the following lemma it is not so easy to get a statement about already assigned nodes from it. First we need to show that in each iteration trees don't grow too much. Notice the $(G^{(i+1)} \setminus G^{(i+3)})$ is the set of nodes that gets removed in the next iteration.

Lemma 28. *For any free node w for all even iterations $i > 1$. For all nodes $v \in (G^{(i+1)} \setminus G^{(i+3)}) \cap T^{(i+3)}(w)$, v has distance less than $3b$ from w.*

Proof. Let $R^{(i+3)} := G^{(i+1)} \setminus G^{(i+3)}$ be the set of all nodes that were first assigned a layer in iteration $i+2$. Let $v \in R^{(i+3)} \cap T^{(i+3)}(w)$. As a result there must be an oriented path from w to v. Since v was free before iteration $i+2$ and therefore was also an orphan at that point in time, such a path must consist completely of nodes in $R^{(i+3)}$. Hence, by giving an upper bound on the longest such oriented path in $R^{(i+3)}$, we can prove the lemma.

At the beginning of iteration $i+2$ all nodes in $R^{(i+3)}$ are free. Edges are only oriented in Lines 8, 19, 20 and 24. Since Line 8 only happens before iteration 2, it does not affect $R^{(i+3)}$. After Line 19 and 20 the longest oriented path in $R^{(k+3)}$ has length at most the length of one of those paths mentioned in these lines. Since components in $V(P') \setminus Z$ have length in $[\ell, 2\ell]$ these paths have at most length 2ℓ. So after these lines any oriented path consisting of nodes from $R^{(i+3)}$ has length at most 2ℓ. What remains is Line 24. Consider some oriented path of nodes of $R^{(i+3)}$ of length at most 2ℓ. In each execution of Line 24 we extend this path by at most 1. So after the γ iterations of this line, the longest oriented path consisting of nodes from $R^{(i+3)}$ has length at most $2\ell + \gamma = 2\ell + \ell + 3 = 3\ell + 3 < 3\ell + 6 = 3b$. □

Ok now we are ready to bound the quality of an already assigned node. In the proof Lemma 27 we will do an inductive argument so for now lets assume it already is true. Intuitively what we will do is ignore all of the nodes that were added in the last iteration (corresponding to the first three layer sets) and then get a bound on everything underneath by applying Lemma 27.

Lemma 29. *Suppose Lemma 27 holds. Suppose w is some already assigned node during iteration i. Let k be the last iteration in which w was still a free node. Then*

$$q^{(i+1)}(w) \leq q^{(k+3)}(w)$$
$$\leq |S_0^{(k+3)}(w)| + |S_1^{(k+3)}(w)| + |S_2^{(k+3)}(w)| + \sum_{j=3}^{\lceil (h^{(k+3)}(w)+1)/b \rceil - 1} \lambda^{j-3} |S_j^{(k+3)}(w)|.$$

Proof. Critically we have to specify in which iteration k we apply Lemma 27, in which w was still a free node. The problem is that if we derive some bound on $q^{(k+1)}(w)$ for some iteration k in which w was still free, then in iteration[8] $k+2$ some neighbor v of w might be raked away and $T^{(k+1)}(v)$ might become part of $T^{(k+3)}(w)$. As a result $T^{(k+3)}(w)$ might be larger than $T^{(k+1)}(w)$ and therefore invalidate our bound on $q^{(k+1)}(w)$, since $q^{(k+3)}(w) > q^{(k+1)}(w)$ might be true. The solution to this is the fact that once w is assigned a layer, the subtree of assigned nodes hanging from w can no longer grow[9] and therefore the quality of w can no longer increase. As a result a bound on the quality that holds when w is assigned a layer, will always stay true afterwards. So we choose k to be last iteration in which w was still a free node and show that in that last iteration the quality didn't grow too much.

If w is assigned in iteration $k = 2$ or before, then by Lemma 28 the subtree hanging from w must be of height strictly less than $3b$. Therefore the quality is bounded by

$$q^{(i+1)}(w) = |S_0^{(i+1)}(w)| + |S_1^{(i+1)}(w)| + |S_2^{(i+1)}(w)|.$$

[8]Remember that iteration $k+2$ is the iteration following iteration k, because iterations go in increments of 2.

[9]This is true, because w will no longer be a free node and edges are only ever oriented from free nodes towards nodes that were just assigned a layer, so no new paths from w to other nodes can be created, after w was assigned a layer.

As this is a stronger statement than what we will derive later, we can freely assume that w was not assigned in the first iteration. Let iteration k be the last iteration such that w was a free node at the end of iteration k. This in particular implies that $k \leq i-2$ and that w is assigned a layer in iteration $k+2 \leq i$. So if we give a bound on $q^{(k+3)}(w)$ then this bound still holds in iteration i, i.e., for $q^{(i+1)}(w)$. (Note that, once a node w is assigned, the definition of $q(w)$ ensures that the quality of w cannot increase since the reassignments of assigned nodes to new layers in Algorithm 2 always assign the nodes to higher layers than used before.)

To give a bound on $q^{(k+3)}(w)$ we first need to understand how $T^{(k+3)}(w)$ looks like. Clearly $T^{(k+1)}(w) \subseteq T^{(k+3)}(w)$, but there might also be some additional nodes that were free nodes after iteration k that now have joined w's subtree of assigned nodes. These nodes then also come with their own subtrees. As a result these nodes would now also be in $T^{(k+3)}(w)$. Let us call the set of these nodes

$$V_w^{(k+1)} := \{v \in G^{(k+1)} \mid v \in T^{(k+3)}(w)\}.$$

We note that $v \in T^{(k+3)}(w)$ implies $v \notin G^{(k+3)}$, just from the definition of subtrees of assigned nodes. Now we can express $T^{(k+3)}(w)$ via

$$T^{(k+3)}(w) = T^{(k+1)}(w) \cup \bigcup_{v \in V_w^{(k+1)}} T^{(k+1)}(v).$$

Since all $v \in V_w^{(k+1)}$ are also contained in $G^{(k+1)}$ we can apply Lemma 27 and we have

$$q^{(k+1)}(v) \leq \sum_{j=0}^{\lceil (h^{(k+1)}(v)+1)/b \rceil - 1} \lambda^j |S_j(v)|.$$

To make things precise we need to relate the height of these trees rooted at such a v to the height of the tree of w. Notice that for any node $u \in T^{(k+1)}(v)$ its distance to w is exactly its distance to v plus the distance from v to w, i.e.,

$$\operatorname{dist}(u, w) = \operatorname{dist}(u, v) + \operatorname{dist}(v, w).$$

By using Lemma 28 we get that for any node $v \in V_w^{(k+1)}$ the distance to w is less than $3b$. As a result, we get that for any node $u \in T^{(k+1)}(v)$ for any $v \in V_w^{(k+1)}$

$$\operatorname{dist}(u, w) = \operatorname{dist}(u, v) + \operatorname{dist}(v, w) \leq \operatorname{dist}(u, v) + 3b.$$

If u was in $S_j^{(k+1)}(v)$ and the distance from v to w were actually exactly $3b$, then u would be in $S_{j-3}^{(k+3)}(w)$. Now since

$$q^{(k+1)}(v) \leq \sum_{j=0}^{\lceil (h^{(k+1)}(v)+1)/b \rceil - 1} \lambda^j |S_j^{(k+1)}(v)|$$

and

$$T^{(k+3)}(w) = T^{(k+1)}(w) \cup \bigcup_{v \in V_w^{(k+1)}} T^{(k+1)}(v),$$

we get that

$$q^{(k+3)}(w) \leq q^{(k+1)}(w) + \sum_{v \in V_w^{(k+1)}} q^{(k+1)}(v)$$

$$\leq \sum_{j=0}^{\lceil (h^{(k+1)}(w)+1)/b \rceil -1} \lambda^j |S_j^{(k+1)}(w)| + \sum_{v \in V_w^{(k+1)}} \sum_{j=0}^{\lceil (h^{(k+1)}(v)+1)/b \rceil -1} \lambda^j |S_j^{(k+1)}(v)|$$

Now again we look at $S_j^{(k+1)}(v)$ for some arbitrary j. We already know that only λ^j of all of these nodes contribute to $q^{(k+3)}(w)$. We now want to express them with respect to the layers $S_0^{(k+3)}(w), \ldots$
We want to relate everything to the layer sets of w

$$q^{(k+3)}(w) \leq \sum_{j=0}^{\lceil (h^{(k+1)}(w)+1)/b \rceil -1} \lambda^j |S_j^{(k+1)}(w)| + \sum_{v \in V_w^{(k+1)}} \sum_{j=0}^{\lceil (h^{(k+1)}(v)+1)/b \rceil -1} \lambda^j |S_j^{(k+1)}(v)|$$

$$\leq |S_0^{(k+3)}(w)| + |S_1^{(k+3)}(w)| + |S_2^{(k+3)}(w)| + \sum_{j=3}^{\lceil (h^{(k+3)}(w)+1)/b \rceil -1} \lambda^{j-3} |S_j^{(k+3)}(w)|.$$

We argue that the second inequality holds. For any node $u \in S_j^{(k+1)}(v)$, we get that its distance from w is at most $3b$ larger than its distance to v. So the farthest layer $S_0^{(k+3)}(w), \ldots$ it could be in would be $S_{j+3}^{(k+3)}(w)$ so if we only want to show that an λ^j fraction of nodes in $S_{j+3}^{(k+3)}$[10] contribute to the quality of $q^{(k+3)}(w)$ we can simply assume the worst case and say that v has this distance $3b$ from w. Because if the distance was smaller, it would only mean that some of the nodes from $S_j^{(k+1)}(v)$ end up in layers before $S_{j+3}^{(k+3)}$ and hence have a bigger coefficient.

So now, as we argued before, since this bound holds in iteration $k+2$ in which w was already assigned a layer, this bound will also hold for later iterations. So we get

$$q^{(i+1)}(w) \leq q^{(k+3)}(w)$$

$$\leq |S_0^{(k+3)}(w)| + |S_1^{(k+3)}(w)| + |S_2^{(k+3)}(w)| + \sum_{j=3}^{\lceil (h^{(k+3)}(w)+1)/b \rceil -1} \lambda^{j-3} |S_j^{(k+3)}(w)|.$$

□

Proof of Lemma 27. We will prove the claim by induction on $h^{(i+1)}(r)$, the height of the assigned subtree $T^{(i+1)}(r)$.
Base Case: $0 \leq h(r) < b$. For $h(r) < b$, the set $S_0(r)$ is just the entire tree $T(r)$, so the statement is trivially true for any $0 < \lambda < 1$.
Induction step: Now we want to prove the statement for some tree with height $h^{(i+1)}(r)$, so let the statement be true for all trees that have height at most $h^{(i+1)}(r) - b$.

We will use the induction hypothesis on all of the nodes in $C_b(r)$. Notice that this is valid, since their heights can be at most $h(r) - b$. Let $w \in C_b(r)$ be one such node. By invoking Lemma 29 we get

$$q^{(i+1)}(w) \leq q^{(k+3)}(w)$$

$$\leq |S_0^{(k+3)}(w)| + |S_1^{(k+3)}(w)| + |S_2^{(k+3)}(w)| + \sum_{j=3}^{\lceil (h^{(k+3)}(w)+1)/b \rceil -1} \lambda^{j-3} |S_j^{(k+3)}(w)|.$$

[10]respectively a λ^{j-3} fraction of nodes in $S_j^{(k+3)}$

We note that this is not a cyclic argumentation, as by the inductive hypothesis Lemma 27 already holds for w and all nodes in its subtree (Critically all of the nodes in the set $V_w^{(k+1)}$). We get such a bound for every node $w \in C_b(r)$. To keep things clearer we will refer to the iteration k which we used for the induction hypothesis as k_w for a specific w, as these iterations might be different for different w. So for every $w \in C_b(r)$ we get

$$q^{(i+1)}(w) \le q^{(k_w+3)}(w)$$

$$\le |S_0^{(k_w+3)}(w)| + |S_1^{(k_w+3)}(w)| + |S_2^{(k_w+3)}(w)| + \sum_{j=3}^{\lceil (h^{(k_w+3)}(w)+1)/b \rceil -1} \lambda^{j-3} |S_j^{(k_w+3)}(w)|$$

$$= \sum_{j=0}^{2} |S_j^{(k_w+3)}(w)| + \sum_{j=3}^{\lceil (h^{(k_w+3)}(w)+1)/b \rceil -1} \lambda^{j-3} |S_j^{(k_w+3)}(w)|$$

Critically, we get bounds on their quality during iteration $k_w + 2$ in which they were just assigned a layer, which must have happened before any node was promoted in Line 32. As a result, we get that no node that was at distance $2b - 1$ from r that was promoted, is accounted for in these bounds on the qualities. This is because the nodes we have used the induction hypothesis on are at distance exactly b from r and at the point at which they were free nodes, they had only promoted nodes at distance exactly b from themselves, so of distance strictly larger than $2b - 1$ with respect to r. And at the point in time at which our bounds on $q^{(i+1)}(w)$ were derived, we did not yet consider any node being promoted in iteration $k_w + 2$. We will denote by $q'(r)$ the quality of r at this point in time, i.e., at a point in time before considering any node at distance $2b - 1$ from r as promoted. We get

$$q'(r) \le |S_0^{(i+1)}(r)| + \sum_{w \in C_b(r)} q^{(k_w+3)}(w)$$

$$\le |S_0^{(i+1)}(r)| + \sum_{w \in C_b(r)} \left(\sum_{j=0}^{2} |S_j^{(k_w+3)}(w)| + \sum_{j=3}^{\lceil (h^{(k_w+3)}(w)+1)/b \rceil -1} \lambda^{j-3} |S_j^{(k_w+3)}(w)| \right).$$

Since $S_j^{(k_w+3)}(w)$ will not change anymore, the set is equal to the set $S_j^{(i+1)}(w)$. By then also observing the fact that

$$\bigcup_{w \in C_b(r)} S_j^{(i+1)}(w) = S_{j+1}^{(i+1)}(r),$$

we obtain

$$q'(r) \le \sum_{j=0}^{3} |S_j^{(i+1)}(r)| + \sum_{j=4}^{\lceil (h^{(i+1)}(r)+1)/b \rceil -1} \lambda^{j-4} |S_j^{(i+1)}(r)|.$$

Again this holds before any node at distance $2b - 1$ from r was promoted. Now we have to differentiate between two cases: either the v^* computed by r was promoted or not. If v^* is promoted, we get by Lemma 25 that $q'(v^*)$ is some constant fraction of $q'(r)$. Now since node v^* is at distance exactly b to r, the nodes that account for $q'(v^*)$ are not yet included in the bounds on $q'(r)$, hence $q(r) = q'(r) - q'(v^*)$. In the other case, where the condition to promote v^* does not hold, we get by Lemma 26 that some x exists such that $q'(x)$ is a constant fraction of $q'(v^*)$ and, by extension, also a constant fraction of $q'(r)$. Furthermore Lemma 26 gives us that this x must be in the first $2b - 1$ layers and therefore also $q(r) \le q'(r) - q'(x)$ would hold.

By Lemmas 25 and 26 we know that the fraction of $q'(r)$ that will be subtracted is smaller in the second case, where we just have that some x exists, that was promoted. So w.l.o.g. we can just consider the second case. By Lemmas 25 and 26 we obtain

$$q'(x) \geq \frac{q'(v^*)}{2\Delta^b} \geq \frac{q'(r)}{4\Delta^{2b}},$$

which implies

$$q^{(i+1)}(r) \leq q'(r) - q'(x) \leq q'(r) - \frac{q'(r)}{4\Delta^{2b}} = (1 - \frac{1}{4\Delta^{2b}})q'(r).$$

By choosing $\lambda^4 = (1 - \frac{1}{4\Delta^{2b}})$, we obtain

$$\begin{aligned} q^{(i+1)}(r) &\leq (1 - \frac{1}{4\Delta^{2l}})q'(r) \\ &= \lambda^4 \left(\sum_{j=0}^{3} |S_j^{(i+1)}(r)| + \sum_{j=4}^{\lceil (h^{(i+1)}(r)+1)/b \rceil - 1} \lambda^{j-4} |S_j^{(i+1)}(r)|. \right) \\ &\leq \sum_{j=0}^{\lceil (h^{(i+1)}(r)+1)/b \rceil - 1} \lambda^{j} |S_j^{(i+1)}(r)|. \end{aligned}$$

□

5.3 Distributed Algorithm and Node Averaged Complexity

In this section, we will describe how we implement Algorithm 2 in a distributed manner and how we will use it to design an algorithm $\mathcal{A}$ that solves the given LCL problem Π, and we will prove an upper bound of $O(\log^* n)$ for the node-averaged complexity of the latter algorithm. We start by describing our distributed implementation of Algorithm 2. For the remainder of the section, set $s := 10\ell$.

Distributed implementation. The computation of ℓ from Π in Line 1 of Algorithm 2 can be performed by every node without any communication. Next, the nodes compute a distance-s coloring with a constant number of colors. Since Δ and ℓ are constant, this can be done in worst-case complexity $O(\log^* n)$, e.g., by computing a $(\Delta(G^s) + 1)$-coloring of the power graph G^s, using the algorithm of Barenboim, Elkin and Kuhn [13] (that computes a $(\Delta + 1)$-coloring in $O(\log^* n + \Delta)$ rounds). Then, the nodes execute Lines 6–8, which they can do by executing γ rounds of a simple rake operation (including orienting edges if required). It remains to consider the for loop in Lines 11–34.

The computation of the subset Z in Line 15 can be performed in a constant number of rounds by iterating through the color classes of the computed distance coloring and in each iteration adding a node to Z if it does not cut off a subpath of P' of length $< \ell$ (which can be determined in ℓ rounds per iteration). Note that no path of length $2\ell + 1$ can remain in the graph induced by $V(P') \setminus Z$ since the middle node of this path would have been added to Z during the iteration of its color class. Analogously, no path of length $> 2\ell + 1$ can remain after the computation of Z.

As additionally each of the γ rake operations (Line 23) can be performed in 1 round, all other operations require only seeing up to distance linear in $|P'|$, b, or $|P''|$ (which are all constants), and the nodes and paths in Lines 12 and 26, respectively, can be processed in parallel, we obtain the following lemma.

Lemma 30. *Assume a distance-s coloring with a constant number of colors is given. Then iteration i of Algorithm 2 can be executed in a constant number of rounds, for each even positive integer i.*

Note that a node can determine in constant time whether it should take part in a certain operation as this only depends on the node's constant-hop neighborhood.

Next we will describe our algorithm $\mathcal{A}$ for solving a given LCL problem Π.

Algorithm for Π. In order to describe our algorithm $\mathcal{A}$ for solving Π with a small node-average complexity, we first describe an algorithm $\mathcal{A}'$ that is not optimized for the node-averaged setting and then explain how to tweak $\mathcal{A}'$ to obtain $\mathcal{A}$.

Algorithm $\mathcal{A}'$ proceeds as follows. Use Algorithm 2 to compute a (γ, ℓ, L)-decomposition, where the values of γ and ℓ depend on Π, and $L \in O(\log n)$ (due to Corollary 17). Then execute the generic algorithm from Section 4.2 using the computed (γ, ℓ, L)-decomposition. As explained in Section 4.2, the generic algorithm produces a correct output for Π given *any* (γ, ℓ, L)-decomposition.

In order to turn $\mathcal{A}'$ into an algorithm $\mathcal{A}$ with small node-averaged complexity, we simply let each node start executing the steps in the generic algorithm as soon as the partial decomposition computed so far by Algorithm 2 provides all necessary information. Additionally, when having determined the output labels for all incident edges, each node immediately terminates.

From the description of the generic algorithm provided in Section 4.2, it follows precisely what information a node needs in order to execute the two steps in the generic algorithm: choosing label-sets for incident edges and choosing labels for incident edges. In particular, the description of the generic algorithm implies the following:

1. Consider a layer computed in the (γ, ℓ, L)-decomposition. At most $2\ell + 1$ rounds after all nodes in all smaller layers (according to the total order given in Definition 8) have chosen label-sets for all incident edges (or earlier), all nodes in the considered layer know the label-set for each incident edge. If there is no smaller layer, all nodes in the considered layer know their label-sets after the first round.

2. If a node is a local maximum and knows its incident label-sets, it can immediately output labels for all incident edges. If a node is in a rake layer (i.e., in a layer of the form $V_{i,j}^R$) and knows its incident label-sets, then it can output labels for all of its incident (so far unlabeled) edges one round after all incident edges of its parent (if it has one) have an output label. If a node is in a compress layer (i.e., in a layer of the form V_i^C) and each node in its path P in the layer knows its incident label-sets, then each node in the path can output labels for all of their incident (so far unlabeled) edges $2\ell + 1$ rounds after all incident edges of the parents of the two endpoints of the path (if they have one) have an output label.

These two properties enable us to prove the following lemma.

Lemma 31. *Assume a distance-s coloring with a constant number of colors is given. Then there exists an integer constant t such that the following holds: if a node v becomes a local maximum in iteration i of Algorithm 2, then the entire tree $T^{(i)}(v)$ will have terminated after ti rounds in $\mathcal{A}$.*

Proof. Let v be a node that becomes a local maximum in iteration i of Algorithm 2. As the design of Algorithm 2 guarantees that there are at most $2i\gamma$ layers below the layer v is assigned to, Property 1 in the above discussion implies that v knows the label-sets for its incident edges latest after round $5i\gamma\ell$, and in turn Property 2 together with Lemma 22 implies that after another at most $5i\gamma\ell$ rounds all nodes in $T^{(i)}(v)$ have chosen output labels for their incident edges and terminated. Now, set t to be the sum of $10\gamma\ell$ and the constant from Lemma 30. □

To make the runtime analysis a bit cleaner, we are going to mark all nodes in $T^{(i)}(v)$, once v becomes a local maximum. We emphasize that this is solely for the purpose of the analysis and this does not change the algorithm at all. More specifically, once any node v becomes a local maximum, all of the nodes in $T^{(i)}(v)$ become marked instantly (in 0 rounds). We obtain the following corollary from Lemma 31.

Corollary 32. *Assume a distance-s coloring with a constant number of colors is given. If a node v becomes marked in iteration i, then v will have terminated in round ti, where t is the constant from Lemma 31.*

Now we use this result to prove that a large amount of nodes are in subtrees of local maxima after a reasonable amount of time. This will be sufficient to prove that the node-averaged complexity is constant, without the time for the input coloring.

Lemma 33. *There exists a constant $0 < \sigma < 1$, such that for every iteration $i \geq 10$ of Algorithm 2 at most $2\Delta^b n\sigma^i$ nodes are not marked.*

Proof. Let F be the set of all marked nodes after iteration i. We want to upper bound the size of $V \setminus F$. By Lemma 21, we get a decomposition of $V(G)$ into

$$V(G) = \left(\bigcup_{v \in G^{(i+1)}} T^{(i+1)}(v) \right) \cup \left(\bigcup_{v \in N^{(i+1)}} T^{(i+1)}(v) \right).$$

For simplicity, we will define the left part as R_i and the right part as C_i, i.e.,

$$R_i := \bigcup_{v \in G^{(i+1)}} T^{(i+1)}(v),\ C_i := \bigcup_{v \in N^{(i+1)}} T^{(i+1)}(v).$$

We obtain that

$$V \setminus F = (R_i \cup C_i) \setminus F = (C_i \setminus F) \cup (R_i \setminus F).$$

So we now need to bound the size of these two sets. To bound the size of $C_i \setminus F$, we observe that since all nodes in $N^{(i+1)}$ are marked in iteration i, all nodes in C_i will be marked. So $C_i \setminus F = \emptyset$ and we obtain

$$|C_i \setminus F| = 0.$$

For the nodes in R_i we will have to put a bit more effort into it. If we look at a tree $T^{(i+1)}(r)$ without any of the already marked nodes, we obtain exactly $H(v)$ from Definition 11. This is because any marked node is in a descendant of some local maximum. Therefore such a node cannot be reached by a path that satisfies the requirements for $H(v)$. So we get that $|T^{(i+1)}(r) \setminus F| = q^{(i+1)}(r)$, which implies

$$|R_i \setminus F| = |\bigcup_{r \in G^{(i+1)}} \left(T^{(i+1)}(r) \setminus F \right)| = \sum_{r \in G^{(i+1)}} q^{(i+1)}(r).$$

Now by applying Lemma 27, we get that

$$|R_i \setminus F| = \sum_{r \in G^{(i+1)}} q^{(i+1)}(r) \leq \sum_{j=0}^{\lceil (h^{(i+1)}(r)+1)/b \rceil - 1} \lambda^j |S_j^{(i+1)}(r)|$$

By noticing that $S_j(r)$ is empty for $jb > h^{(i+1)}(r) + 1$ and defining $h = \max\{h^{(i+1)}(r) \mid r \in G^{(i+1)}\}$ we can reformulate the expression as

$$\sum_{r\in G^{(i+1)}} \sum_{j=0}^{\lceil (h^{(i+1)}(r)+1)/b\rceil -1} \lambda^j |S_j(r)| = \sum_{r\in G^{(i+1)}} \sum_{j=0}^{\lceil (h+1)/b\rceil -1} \lambda^j |S_j(r)|.$$

Observe that the maximum height h is upper bounded by $10i\gamma$ (as, e.g., follows from the proof of Lemma 28) which is in turn upper bounded by ti where t is the constant from Lemma 31 and Corollary 32. Hence, we obtain

$$\sum_{r\in G^{(i+1)}} \sum_{j=0}^{\lceil (h+1)/b\rceil -1} \lambda^j |S_j(r)| \le \sum_{r\in G^{(i+1)}} \sum_{j=0}^{\lceil (ti+1)/b\rceil -1} \lambda^j |S_j(r)|.$$

By splitting at some β fraction of the height, which we will fix later, we obtain

$$\begin{aligned}\sum_{r\in G^{(i+1)}} \sum_{j=0}^{\lceil (ti+1)/b\rceil -1} \lambda^j |S_j(r)| &\le \sum_{r\in G^{(i+1)}} \left(\sum_{j=0}^{\lfloor \beta ti/b\rfloor} \lambda^j |S_j(r)| + \sum_{j=\lceil \beta ti/b\rceil}^{\lceil (ti+1)/b\rceil -1} \lambda^j |S_j(r)| \right) \\ &\le \sum_{r\in G^{(i+1)}} \left(\sum_{j=0}^{\lfloor \beta ti/b\rfloor} 1\cdot |S_j(r)| + \sum_{j=\lceil \beta ti/b\rceil}^{\lceil (ti+1)/b\rceil -1} \lambda^{\lceil \beta ti/b\rceil} |S_j(r)| \right).\end{aligned}$$

Now we want to upper bound the number of nodes that are in the first sum. To this end, we notice that in a tree of height βti there can be at most $\Delta^{\beta ti}$ nodes. Furthermore, $|G^{(i+1)}| \le n\varepsilon^i$ by Lemma 16, which implies

$$\begin{aligned}\sum_{r\in G^{(i+1)}} \sum_{j=0}^{\lfloor \beta ti/b\rfloor} |S_j(r)| &\le \sum_{r\in G^{(i+1)}} \Delta^{\beta ti+b} \le |G^{(i+1)}|\Delta^{\beta ti+b} \\ &\le n\varepsilon^i \Delta^{\beta ti+b} = \Delta^b n \cdot \exp\left(i\left(\beta t \ln(\Delta) - \ln(1/\varepsilon)\right)\right)\end{aligned}$$

Now by choosing $\beta = \frac{\ln(1/\varepsilon)}{2t\ln(\Delta)}$, we obtain

$$\Delta^b n \cdot \exp\left(i\left(\beta t \ln(\Delta) - \ln(1/\varepsilon)\right)\right) = \Delta^b n \cdot \exp\left(-i \cdot \frac{1}{2} \cdot \ln(1/\varepsilon)\right) = \Delta^b n \cdot \left(\sqrt{\varepsilon}\right)^i.$$

So the bound we get is

$$\sum_{r\in G^{(i+1)}} \sum_{j=0}^{\lfloor \beta ti/b\rfloor} |S_j(r)| \le \Delta^b n \cdot \left(\sqrt{\varepsilon}\right)^i.$$

Now to bound the size of the second sum, we observe that

$$\sum_{r\in G^{(i+1)}} \sum_{j=\lceil \beta ti/b\rceil}^{\lceil (ti+1)/b\rceil -1} \lambda^{\lceil \beta ti/b\rceil} |S_j(r)| = \lambda^{\lceil \beta ti/b\rceil} \cdot \sum_{r\in G^{(i+1)}} \sum_{j=\lceil \beta ti/b\rceil}^{\lceil (ti+1)/b\rceil -1} |S_j(r)| \le n \cdot \left(\lambda^{\beta t/b}\right)^i.$$

Combining the two obtained inequalities yields

$$|R_i \setminus F| \le \sum_{r\in G^{(i+1)}} \left(\sum_{j=1}^{\lfloor \beta ti/b\rfloor} |S_j(r)| + \sum_{j=\lceil \beta ti/b\rceil}^{\lceil (ti+1)/b\rceil -1} \lambda^{\lceil \beta ti/b\rceil} |S_j(r)| \right) \le \Delta^b n \cdot \left(\sqrt{\varepsilon}\right)^i + n \cdot \left(\lambda^{\beta t/b}\right)^i.$$

It follows that

$$|V \setminus F| = |(R_i \cup C_i) \setminus F| = |((C_i \setminus F) \cup (R_i \setminus F)| \le 0 + \Delta^b n \cdot \left(\sqrt{\varepsilon}\right)^i + n \cdot \left(\lambda^{\beta\gamma/l}\right)^i.$$

By choosing $\sigma = \max\{\lambda^{\beta\gamma/l}, \sqrt{\varepsilon}\}$, we obtain

$$|V \setminus F| \le 2\Delta^b n \sigma^i$$

□

We obtain the following lemma.

Lemma 34. *On average, nodes become marked in $O(1)$ iterations.*

Proof. By Lemma 33, we get that for each (even) iteration $i > 10$, only $2\Delta^b n\sigma^i$ nodes are not marked. Note that the number of iterations of a node v is therefore at most 5 plus the total number of even iterations $i > 10$ in which that node has not terminated. We can therefore upper bound the total number of iterations of all nodes as $5n$ plus the sum over $2\Delta^b n \cdot \sigma^i$ for all even $i > 10$. This sum can be upper bounded by summing over both the even and odd i and to get the average number of iterations per node we divide by n. We therefore get an upper bound of

$$5 + \frac{1}{n}\sum_{i=12}^{\infty} 2\Delta^b n \cdot \sigma^i \le 5 + 2\Delta^b \sum_{i=1}^{\infty} \sigma^i = 5 + \frac{2\Delta^b}{1-\sigma} \in O(1).$$

on the average number of iterations per node. □

Then using this lemma together with Corollary 32 we get that an average node terminates after a constant number of rounds. However, we still have to pay for the input distance coloring which takes $O(\log^* n)$, as discussed in the beginning of the section. So by first computing this input coloring and then running the algorithm, we obtain a total node-averaged complexity of $O(\log^* n)$, proving Theorem 1.

6 Improved Algorithms in the Polynomial Regime

In this section we show that, for infinitely many LCL problems with polynomial worst-case complexity, we can improve their node-averaged complexity. More precisely, in this section we show that, for a class of problems with worst-case complexity $\Theta(n^{1/k})$, we can provide an algorithm with node-averaged complexity $O(n^{1/(2^k-1)})$. As we show in Section 7, this complexity is almost tight, since for all problems with worst-case complexity $\Theta(n^{1/k})$ we can show a lower bound of $\widetilde{\Omega}(n^{1/(2^k-1)})$.

6.1 The Hierarchical $2\frac{1}{2}$-Coloring Problems

We now define a class of problems, already presented in [19], called hierarchical $2\frac{1}{2}$-coloring, that is parametrized by an integer $k \in \mathbb{Z}^+$. It has been shown in [19] that the problem with parameter k has worst-case complexity $\Theta(n^{1/k})$. We now give a formal definition of this class of problems and then we provide some intuition.

The set of input labels is $\Sigma_{\text{in}} = \emptyset$. The set of output labels contains four possible labels, that is, $\Sigma_{\text{out}} = \{W, B, E, D\}$, and these labels stand for *white*, *black*, *exempt*, and *decline*. Each node has a level in $\{1, \ldots, k+1\}$, that can be computed in constant time, and the constraints of the nodes depend on the level that they have. The level of a node is computed as follows.

1. Let $i \leftarrow 1$.
2. Let V_i be the set of nodes of degree at most 2 in the remaining tree. Nodes in V_i are of level i. Nodes in V_i are removed from the tree.
3. Let $i \leftarrow i + 1$. If $i \leq k$, continue from step 2.
4. Remaining nodes are of level $k + 1$.

Each node must output a single label in Σ_{out}, and based on their level, they must satisfy the following local constraints.

- No node of level 1 can be labeled E.
- All nodes of level $k + 1$ must be labeled E.
- Any node of level $2 \leq i \leq k$ is labeled E iff it is adjacent to a lower level node labeled W, B, or E.
- Any node of level $1 \leq i \leq k$ that is labeled W (resp. B) has no neighbors of level i labeled B (resp. W) or D. In other words, W and B are colors, and nodes of the same color cannot be neighbors in the same level.
- Nodes of level k cannot be labeled D.

This problem can be expressed as a standard LCL (a type of LCLs defined in Appendix C) by setting the checkability radius r to be $O(k)$, since in $O(k)$ rounds a node can determine its level and hence which constraints apply.

Some intuition on these problems. In order to have a bit of intuition on this class of problems, let us consider the case when $k = 2$, for which the worst-case complexity is $\Theta(\sqrt{n})$. Let us focus on nodes of levels 1 and 2 (note that these are the only nodes that matter, since nodes of level 3 will blindly output E). Notice that, by the definition of the level of a node, nodes of the same level form paths. Let Q be a path of level-1 nodes. By the definition of the LCL problems described above, Q must either be 2-colored by using the labels W and B, or all nodes in Q must be labeled D. Now consider a path P of level-2 nodes. By the definition of the LCL problems, a node in P must be labeled E if and only if it has a level-1 neighbor labeled with W or B. On the other hand, the subpaths of P induced by nodes having a level-1 neighbor labeled D must be 2-colored by using the labels W and B.

On the lower bound side, a worst-case instance for this LCL with parameter $k = 2$ consists of a path P of length $\Theta(\sqrt{n})$, where to each node v_j of P is attached a path Q_j of length $\Theta(\sqrt{n})$. If an algorithm performs a 2-coloring of any of the Q_j paths, then it needs to spend $\Omega(\sqrt{n})$ rounds. Otherwise, if none of the Q_j paths gets 2-colored, then any correct algorithm must 2-color P, spending $\Omega(\sqrt{n})$ rounds.

On the upper bound side, this LCL with parameter $k = 2$ can be solved in $O(\sqrt{n})$ rounds in the following way. First, level-1 nodes spend $O(\sqrt{n})$ rounds to check if the path that they are in has length $O(\sqrt{n})$: if yes, then the path gets 2-colored with labels W and B; if no, the path gets labeled with D. If a level-2 node has a level-1 neighbor that is colored (i.e., it is labeled W or B), then it outputs E. Finally, it is possible to prove that the subpaths induced by nodes of level 2 that have a level-1 neighbor labeled D must be of length $O(\sqrt{n})$, hence these subpaths can be 2-colored in $O(\sqrt{n})$ rounds.

6.2 Better Node-Averaged Complexity

We now show that, for the class of LCL problems described in Section 6.1, we can obtain a better node-averaged complexity. The algorithm is similar to the one presented in [16] for the worst-case complexity, but it is modified to obtain a better node-averaged complexity (in Section 7 we show that this algorithm is tight up to a $\log n$ factor).

Theorem 35. *The node-averaged complexity of the hierarchical $2\frac{1}{2}$-coloring problem with parameter k is $O(n^{1/(2^k-1)})$.*

Proof. At first, all nodes spend $O(1)$ rounds to compute their level. Nodes of level $k+1$ output E. Then, the algorithm proceeds in phases, for i in $1, \ldots, k$. In phase i, all nodes of level i get a label, and hence let us assume that all nodes of levels $1, \ldots, i-1$ already have a label, and let us focus on level-i nodes.

Consider a node v of level i. Node v proceeds as follows. If v has a neighbor from lower levels that is labeled W or B then v outputs E. Otherwise, v spends $t_i = c \cdot \gamma_i$ rounds to check the length of the path containing v induced by nodes of level i, for some constant c to be fixed later, and $\gamma_i = n^{2^{i-1}/(2^k-1)}$. If this length is strictly larger than t_i, then v outputs D. Otherwise, all nodes of the path are able to see the whole path, and hence they can output a consistent 2-coloring by using the labels W and B.

The above algorithm correctly solves the problem if we assume that no nodes in level k output D. In the following we show that indeed nodes of level k do not output D, hence showing the correctness of the algorithm, and then we prove a bound on the node-averaged complexity.

In order to do so, we first prove a useful statement. Let S be the set of nodes of level i that do not directly output E at the beginning of phase i. It is possible to assign each node of level $j < i$ to exactly one node in S such that to each node in S are assigned $\Omega(n^{(2^{i-1}-1)/(2^k-1)})$ unique nodes of lower layers. Let v be a node in S. Since $v \in S$, it means that v is connected to a path P of nodes of level $i-1$ of length strictly larger than t_{i-1}, and hence all nodes in P are labeled D. Since, by the definition of the levels, P has at most 2 nodes of higher levels connected to it, then we can charge $t_{i-1}/2$ unique nodes to v. By repeating this reasoning inductively, we obtain that for each node of layer i that does not directly output E, we can assign at least the following amount of nodes:

$$\prod_{j=1}^{i-1} t_j/2 = \Omega\left(\prod_{j=1}^{i-1} \gamma_j\right) = \Omega(n^{\sum_{j=1}^{i-1} 2^{j-1}/(2^k-1)}) = \Omega(n^{(2^{i-1}-1)/(2^k-1)}).$$

Hence, the number of nodes that participate in phase i is at most $O(n^{1-(2^{i-1}-1)/(2^k-1)}) = O(n^{(2^k-2^{i-1})/(2^k-1)})$. This implies that in phase k the number of participating nodes is at most $O(n^{(2^k-2^{k-1})/(2^k-1)}) = O(n^{2^{k-1}/(2^k-1)}) = O(\gamma_k)$, where the hidden constant is inversely proportional to c. Hence, by picking c large enough, we get that in t_k rounds nodes of level k see the whole path and thus no node of level k outputs D, proving the correctness of the algorithm.

The total time spent during phase i is bounded by

$$O(\gamma_i \cdot n^{1-(2^{i-1}-1)/(2^k-1)}) = O(n^{2^{i-1}/(2^k-1)} \cdot n^{1-(2^{i-1}-1)/(2^k-1)}) = O(n^{1+1/(2^k-1)}).$$

Therefore, the average time spent in phase i is bounded by $O(n^{1/(2^k-1)})$. Since this is done for $i \in \{1, \ldots, k\}$, and since $k = O(1)$, then the claim on the node-averaged complexity follows. □

7 Lower Bounds in the Polynomial Regime

In this section we show that any LCL problem that requires polynomial time for worst-case complexity requires polynomial time also for node-averaged complexity. More precisely, we prove the following theorem.

Theorem 36. *Let Π be an LCL problem with worst-case complexity $\Omega(n^{1/k})$ in the* LOCAL *model. The node-averaged complexity of Π in the* LOCAL *model is $\Omega(n^{1/(2^k-1)}/\log n)$.*

In order to prove this theorem, we proceed as follows (throughout this section we will use notions presented in Section 4.2). It is known by [16] that if an LCL problem Π has worst-case complexity $o(n^{1/k})$, then it can actually be solved in $O(n^{1/(k+1)})$ rounds. This statement is proved by showing that it is possible to use an algorithm (possibly randomized) running in $o(n^{1/k})$ rounds to construct a good function $f_{\Pi,k+1}$ (that is, a function $f_{\Pi,k+1}$ that, if used, never creates empty classes), implying (as shown in Section 4.2) the existence of a deterministic algorithm that solves Π and has worst-case complexity $O(n^{1/(k+1)})$. In this section we show that it is possible to construct a good function $f_{\Pi,k+1}$ by starting from an algorithm $\mathcal{A}$ with node-averaged complexity $o(n^{1/(2^k-1)}/\log n)$. By Section 4.2, this implies that if there exists an algorithm with $o(n^{1/(2^k-1)}/\log n)$ node-averaged complexity, then there exists an algorithm with worst-case complexity $O(n^{1/(k+1)})$, implying that any LCL with worst-case complexity $\Omega(n^{1/k})$ has node-averaged complexity at least $\Omega(n^{1/(2^k-1)}/\log n)$. While we will use some ideas already presented in [16], handling an algorithm with only guarantees on its node-averaged complexity arises many (new) issues that we need to tackle.

Our statement will be proved even for the case in which algorithm $\mathcal{A}$ satisfies the weakest possible assumptions (i.e., the assumptions are so relaxed that they are satisfied by any deterministic algorithm, any randomized Las Vegas algorithm, and any randomized Monte Carlo algorithm). The assumptions are the following.

- We assume that $\mathcal{A}$ is a randomized algorithm that is only required to work when the unique IDs of nodes are assigned at random, among all possible valid assignments.
- We assume that $\mathcal{A}$ is an algorithm that fails with probability at most $1/n^c$ for any chosen constant $c \geq 1$.
- We assume that the bound on the node-averaged complexity of $\mathcal{A}$ holds with probability at least $1 - 1/n^c$ for any chosen constant $c \geq 1$.

However, in the following, we will assume that the bound on the node-averaged complexity of $\mathcal{A}$ holds always. In fact, observe that we can always convert an algorithm with node-averaged complexity T that holds with probability at least $1 - 1/n^c$ into an algorithm with node-averaged complexity $O(T)$ that holds always, since, even for $c = 1$, we can safely assume that when the bound does not hold (that happens with probability at most $1/n$), the runtime is anyways bounded by n (since everything can be solved in n rounds in the LOCAL model).

7.1 Good Functions

We now show how, in [19, 16], it is determined whether a good function exists. In the following, by *solver* we mean the algorithm for solving an LCL given a decomposition, presented in Section 4.2.

The high level idea in [19, 16] is that, being that there are a finite amount of possible functions (because the classes of Definition 4 have finite size, and because there is a finite amount of ways to map maximal classes into independent ones), we can test them all by using a centralized algorithm that checks whether a function is good. The key question is how to test whether a function is good, and, on a high level, this is done by constructing all possible label-sets that could possibly appear while running the solver, which depend on the tested function. Crucially, this is a finite amount, and there is a recursive procedure that can generate them.

What actually determines the complexity of a problem. We now provide some more intuition about the function $f_{\Pi,k}$, and about how the existence of this function is related with the complexity of a problem. In [19, 16] it is shown how to determine whether a good function exists (that is, a function that never creates empty classes), and if it exists, how to construct it. Also, it is shown that:

- If a problem is solvable in $O(\log n)$ rounds, then a good function $f_{\Pi,\infty}$ exists.
- If a problem is solvable in $o(n^{1/k})$, then a good function $f_{\Pi,k+1}$ exists.

This implies that, if a problem is solvable in $O(\log n)$ rounds, then we can automatically find a good function that makes the generic algorithm work and solve the problem in $O(\log n)$ rounds, and if a problem is solvable in $o(n^{1/k})$, then we can automatically find a good function that makes the generic algorithm work and solve the problem in $O(n^{1/(k+1)})$ rounds. Also, given a problem Π, it is possible to compute what is the optimal target complexity, that is, we can determine whether the problem can be solved in $O(\log n)$ rounds, and if the answer is negative we can determine the best integer k for which the problem can be solved in $O(n^{1/k})$ rounds.

The intuition about what determines the existence of a good function $f_{\Pi,k}$ is the following: compress paths are something that is difficult to handle, because they require to restrict the label-sets that we propagate up in order to make them "independent". The number of compress layers that we can recursively handle is what determines the complexity of a problem:

- If we can handle an arbitrary amount of compress layers, then we can construct a good function $f_{\Pi,\infty}$, and hence the problem can be solved in $O(\log n)$ rounds.
- If we can handle only a constant amount of compress layers, say $k-1$, then a good function $f_{\Pi,k}$ exists, but $f_{\Pi,k+1}$ does not, and then the complexity of the problem is $\Theta(n^{1/k})$.

Testing procedure. We now present an algorithm that tests whether a function $f_{\Pi,k}$ is good. We call this algorithm *testing procedure*. The procedure depends not only on the function to be tested, but also on a parameter ℓ that, in [19, 16], is shown that it can be determined solely as a function of Π. We observe that this algorithm is well-defined also when $k = \infty$, and in fact this algorithm can be used also to determine whether a problem can be solved with $O(\log n)$ worst-case complexity. The idea of the testing procedure is to keep track of all possible label-sets that one could possibly obtain while running the solver. For each of these label-sets, we also keep track of a subtree (where nodes are also marked with the layers of a decomposition) where, if we run the solver by using the function that we are testing, we would obtain an edge with such a label-set. While this is not necessary for *testing* a function, we will use these trees later for *constructing* a function given an algorithm. We now formally describe the testing procedure and later we will give more intuition on it.

1. Initialize S with all the possible values of the label-set $g(v)$ of v (as defined in Definition 5) that could be obtained when v is a leaf. Note that the possible values are a finite amount that only depends on the amount of input labels of Π. Initialize $\mathcal{R}_1$ with one pair $((\tilde{T}, v), L)$ for each element L in S, where $\tilde{T}$ is a tree composed of 2 nodes $\{u, v\}$ and 1 edge $\{u, v\}$, and $g(v) = L$. Node v is marked as being a rake node of layer 1, while u is marked as being a temporary node.

2. For $i = 1, \dots, k$ do the following. If, at any step, an empty label-set is obtained, then the tested function is not good.

   (a) Do the following in all possible ways: consider x arbitrary elements $((\tilde{T}_j, v_j), L_j)$ of $\mathcal{R}_i$, where $1 \leq j \leq x$ and $1 \leq x \leq \Delta$. Construct the tree T as the union of all trees $\tilde{T}_j$, where all the nodes v_j (note that each node v_j has degree 1) are identified as node v, which, after this process, has degree x in T. Let F_{incoming} be the set of edges connected to v, and let $\mathcal{L}_{\text{incoming}}$ be the label-set assignment given by the sets L_j. The node v is marked as a rake node of layer i. If v has an empty maximal class w.r.t. F_{incoming}, $F_{\text{outgoing}} = \{\}$, and $\mathcal{L}_{\text{incoming}}$, then the tested function is not good.

   (b) Do the following in all possible ways: consider x arbitrary elements $((\tilde{T}_j, v_j), L_j)$ of $\mathcal{R}_i$, where $1 \leq j \leq x$ and $1 \leq x \leq \Delta - 1$. Construct the tree T as the union of all trees $\tilde{T}_j$, where all the nodes v_j are identified as node v. Attach an additional neighbor u to v. Let $F_{\text{outgoing}} = \{\{u, v\}\}$. Let F_{incoming} be the set of edges connected to v, excluding $\{u, v\}$, and let $\mathcal{L}_{\text{incoming}}$ be the label-set assignment given by the sets L_j. The node v is marked as a rake node of layer i, while u is marked as a temporary node. Let $L = g(v)$ (as defined in Definition 5). If L is empty, then the function is not good.

   (c) Repeat the previous two step until nothing new is added to $\mathcal{R}_i$. This must happen, since there are a finite amount of possible label-sets.

   (d) If $i = k$, stop.

   (e) Initialize $\mathcal{C}_i = \emptyset$.

   (f) Do the following in all possible ways: construct a graph starting from a path H of length between ℓ and 2ℓ where we connect nodes of degree 1 to the nodes of H satisfying: (i) all nodes in H have degree at most Δ; (ii) the two endpoints of H have an outgoing edge that connects respectively to nodes u_1 and u_2 that are nodes of degree 1; (iii) all the other edges connecting degree-1 nodes to the nodes of H are incoming for H. Next, replace each incoming edge e and the node of degree 1 connected to it with a tree $\tilde{T}$ of a pair $((\tilde{T}, v), L)$ in $\mathcal{R}_i$, by identifying v with the node of the path connected to e. Different trees can be used for different edges. The nodes u_1 and u_2 are marked as temporary nodes, while the nodes of the path are marked as compress nodes of layer i. Use the function as described in Definition 5 to compute the label-sets L_1 and L_2 of the two endpoints. If L_1 or L_2 is empty, then the function is not good. Otherwise, add the pairs $((H, u_1), L_1)$ (resp. $((H, u_2), L_2)$) to $\mathcal{C}_i$ if no pair with second element L_1 (resp. L_2) is already present. The *representative tree* of $P = (H, F_{\text{incoming}}, F_{\text{outgoing}}, \mathcal{L}_{\text{incoming}})$ is defined as $r(P) = T$.

   (g) Set $\mathcal{R}_{i+1} = \mathcal{R}_i \cup \mathcal{C}_i$. If $\mathcal{R}_{i+1} = \mathcal{R}_i$, stop.

On a high level, the testing procedure does the following. Item 1 computes all possible label-sets that we can get from leaf nodes; leaf nodes are then marked as rake nodes. Item 2a considers all possible ways to assign a label-set to all the incident edges of a node v, and checks, for each case, if there is a way for v to label its incident edges (with a label from the provided label-sets) such that the constraints of node v are satisfied. Node v is marked as a rake node. Item 2b considers all possible ways to assign a label-set to all but one incident edge of a node v; let e be the edge that does not have a label-set assigned. Then, the label-set of e is computed as a function of all the label-sets of the other edges incident to v, and then v is marked as a rake node. In Item 2c the last two steps are repeated until no new label-sets are obtained. In other words, items 2a, 2b,

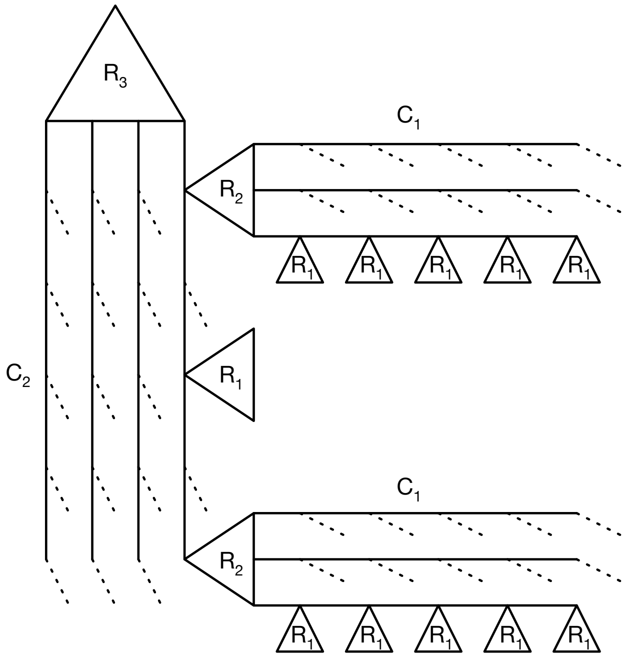

Figure 9: An example of a tree generated by the testing procedure. Triangles marked R_i correspond to trees of constant size containing only nodes marked rake of layer i. Paths marked C_i correspond to nodes marked compress of layer i. At the end of each path, on the side not connected to a triangle, is connected a temporary node, not shown in the picture. The top node of the triangle marked R_3 may be a temporary node.

and 2c generate all possible label-sets that we can get by performing only rakes, that is, before performing the first compress. Finally, Item 2f considers all possible label-sets that we can obtain for the endpoints of a compress path. By repeating recursively the procedure interleaving between rake and compress nodes, and by checking at any point that we do not get empty label-sets, we can test whether a function is good or not. An example of a tree generated by this procedure is shown in Figure 9.

It is possible to prove that the testing procedure generates exactly those label-sets that could possibly be obtained by running the solver [19, 16]. Hence, if empty label-sets are never obtained, then the function can indeed be used to solve a problem. Observe that it is possible to show that all the edges of a generated tree can be oriented such that: all nodes have at most one outgoing edge; any directed path contains layers in non-decreasing order. In a generic tree decomposition, this may be false, but this is not an issue: the obtained label-sets, in this restricted set of cases, are still all the possible ones that could be obtained on an arbitrary tree decomposition.

7.2 Some Useful Ingredients

Before defining the function $f_{\Pi,k+1}$ using algorithm $\mathcal{A}$, we provide some ingredients that will later be useful.

Pumping lemma for trees. In order to prove our statement, we will use a fundamental ingredient provided in [19], that, informally, allows us to take a compress path of some class, and make the path longer while preserving its class.

Lemma 37 (Pumping lemma for trees [19])**.** *Assume we are given an LCL* $\Pi = (\Sigma_{\text{in}}, \Sigma_{\text{out}}, C_W, C_B)$ *in the black-white formalism. Let a* compress path *be a tuple* $(H, F_{\text{incoming}}, F_{\text{outgoing}}, \mathcal{L}_{\text{incoming}})$

satisfying the following:

- *To each node v_i of H are connected d_i neighbors, $u_{i,1}, \dots, u_{i,d_i}$. F_{incoming} is the set of edges connecting the nodes of H with such neighbors.*
- *To each edge e of F_{incoming} is assigned a label-set $L_e \subseteq \Sigma_{\text{out}}$. $\mathcal{L}_{\text{incoming}} = (L_e)_{e \in F_{\text{incoming}}}$ is this assignment.*
- *To each endpoint of H is connected one additional neighbor. F_{outgoing} is the set containing the two edges connecting the endpoints of H to such neighbors.*

Let $(H, F_{\text{incoming}}, F_{\text{outgoing}}, \mathcal{L}_{\text{incoming}})$ be a compress path where H has length x. Let w be an arbitrary integer that is at least x. There exists a constant ℓ such that, if $x \geq \ell$, then it is possible to construct a compress path $(H', F'_{\text{incoming}}, F'_{\text{outgoing}}, \mathcal{L}'_{\text{incoming}})$ satisfying the following:

- *The length of H' is at least w and at most $w + \ell$.*
- *The maximal class of H' w.r.t. Π, F'_{incoming}, F'_{outgoing}, and $\mathcal{L}'_{\text{incoming}}$ is equal to the maximal class of H w.r.t. Π, F_{incoming}, F_{outgoing}, and $\mathcal{L}_{\text{incoming}}$.*
- *The set of label-sets contained in $\mathcal{L}'_{\text{incoming}}$ is equal to the set of label-sets contained in $\mathcal{L}_{\text{incoming}}$.*

Pumping trees. Let T be a tree constructed in the testing procedure. Observe that, in this tree, the nodes are either marked rake, compress, or temporary, and with a layer number. We provide a procedure $\text{pump}(w, T)$ that modifies T to make all compress paths longer. In particular, all compress paths of layer i will be of length between w_i and $w_i + \ell$, where w_i is defined to be $w^{2^{i-1}/(2^k-1)}$. The procedure $\text{pump}(w, T)$ performs the following operation recursively, by starting from compress layer $i = 1$ and going up, and by considering all compress paths present in T.

Let $(H, F_{\text{incoming}}, F_{\text{outgoing}}, \mathcal{L}_{\text{incoming}})$ be a compress path induced by a connected component of nodes marked as compress nodes of layer i. We apply Lemma 37 with parameter w_i and get $(H', F'_{\text{incoming}}, F'_{\text{outgoing}}, \mathcal{L}'_{\text{incoming}})$. We remove all the subtrees that are connected to H via edges in F_{incoming}. We replace H with H'. Let $\{u, v\}$ be an edge in F'_{incoming}, where u is the node of H'. Note that $\{u, v\}$ has a label-set assigned according to $\mathcal{L}'_{\text{incoming}}$. For each edge $\{u, v\}$ in F'_{incoming} (where u is the node of H') we attach a copy of one subtree that was connected to H satisfying that the label-set assigned to the edge connecting the subtree to H is the same as the label-set of the edge $\{u, v\}$. Note that this is possible since the set of the label-set in $\mathcal{L}'_{\text{incoming}}$ is the same as the set of the label-sets in $\mathcal{L}_{\text{incoming}}$, and hence the label-set of $\{u, v\}$ must be a label-set that occurs in some edge in F_{incoming} that connects some subtree to a node of H.

Adding nodes. We now provide a procedure that takes in input a value n and a pumped tree of at most n nodes, and adds nodes to the tree in order to make the amount of nodes to be exactly n. The procedure $\text{add}(n, T)$ is defined as follows. Let j be the maximum compress layer index of the nodes present in T. Let v be a temporary node of T that is connected to a node of compress layer j. We attach a path to v in such a way that the number of nodes becomes exactly n.

Optimal form of an algorithm. It has been shown in [22] that any algorithm can be normalized in a way that nodes do not spend useless rounds of computation. While such a result has been proved on graphs with linearly bounded growth, it can be easily adapted, in the case of LCLs in the black-white formalism, to be more general.

Lemma 38 ([22])**.** *Let A be an algorithm that solves an LCL problem Π with node-averaged complexity T. Then, if a node v runs for t rounds, either there exist at least $t/2 - 1$ nodes in the $t/2$-radius neighborhood of v that run for at least $t/2$ rounds, or v could terminate earlier.*

Proof Sketch. Suppose that in the t-radius neighborhood of v there are at most $t/2-2$ nodes that run for at least $t/2$ rounds. This means that the connected component C induced by v and nodes that still have to terminate after $t/2$ rounds is of size strictly less than $t/2$, and that v in strictly less than t rounds sees the state that all these nodes had at time $t/2$, and of their neighbors. We show that this implies that v can terminate in strictly less than t rounds. Since C is surrounded by nodes that already terminated, then v, solely as a function of its state, the state that the nodes of C had at time $t/2$, and the output of the nodes connected to C, can simulate the execution of A for all nodes of C without additional communication, and obtain an output for all the nodes in C, and hence v can terminate in strictly less than t rounds. $\square$

In the following we will assume $\mathcal{A}$ to be an algorithm for Π that has been normalized according to Lemma 38. We now show that a stronger notion of node-averaged complexity must hold for $\mathcal{A}$, namely that the expected running time of each node is bounded.

Lemma 39. *Let $\mathcal{A}$ be an algorithm with node-averaged complexity at most $c' \cdot n^{1/(2^k-1)}/\log n$, and let $T = \mathrm{add}(n, \mathrm{pump}(w, T'))$, where T' is a tree that has been constructed by the testing procedure and $w = \Theta(n)$.*

Assume that $\mathcal{A}$ is run in T. Then, for each node v in a compress layer i it must hold that the expected running time is at most $c \cdot n^{2^{i-1}/(2^k-1)}/\log n$, for some constant c directly proportional to c', and for all $n \geq n'$ for some constant n'.

Proof. Assume that there exists a node v in a compress layer i with expected running time strictly larger than $t = c \cdot n^{2^{i-1}/(2^k-1)}/\log n$, where the expectation is taken over all possible random bit string assignments and all possible valid ID assignments to the nodes. Let B be the t-radius neighborhood of v. Observe that, by construction (see Figure 9), this neighborhood contains $y = O(t \cdot \prod_{j=1}^{i-1} n^{2^{j-1}/(2^k-1)}) = O(n^{(2^i-1)/(2^k-1)}/\log n)$ nodes. Moreover, for n large enough, this neighborhood does not contain all nodes of T (n' is chosen to be such value of n). This implies that we can make $x = \Omega(n/y)$ copies of B and connect them in such a way that each copy of v has the same view in the constructed tree T and in B. By using Lemma 38, we obtain that, by running $\mathcal{A}$ on B, the sum of the expected running times is strictly larger than $x \cdot (t/2)(t/2-1)$, that, for some constant c'', is at least,

$$c'' \cdot n \cdot \frac{c^2 \cdot n^{2^i/(2^k-1)}/\log^2 n}{c \cdot n^{(2^i-1)/(2^k-1)}/\log n} = \frac{c'' \cdot c \cdot n}{\log n} \cdot n^{(2^i-2^i+1)/(2^k-1)} = c'' \cdot c \cdot n \cdot n^{1/(2^k-1)}/\log n.$$

This implies that the average running time is strictly larger than $c'' \cdot c \cdot n^{1/(2^k-1)}/\log n$. By setting $c = c'/c''$, we get a contradiction on the assumption on the runtime of $\mathcal{A}$. $\square$

7.3 The $f_{\Pi,k+1}$ Function Given $\mathcal{A}$

Function definition. We now show how to define the function $f_{\Pi,k+1}$ by using the given algorithm $\mathcal{A}$ that solves Π with node-averaged complexity $o(n^{1/(2^k-1)}/\log n)$. Recall that the input of the function $f_{\Pi,k+1}$ is a compress path $P = (H, F_{\text{incoming}}, F_{\text{outgoing}}, \mathcal{L}_{\text{incoming}})$, and it is required to produce an independent class for it. As a technicality, let us mention that, for each compress path P, the representative tree $r(P)$ is well defined, since we can define $f_{\Pi,k+1}$ while being tested by the testing procedure.

Let N_f be the maximum size of any graph obtained from the function pump as a function of the parameter w when applied to the graphs constructed in the testing procedure with the function f. Then, let N be the maximum value of N_f, taken over all possible functions f, again as a function of the parameter w. We are now ready to fix the value of the parameter w. We first pick w sufficiently large such that the average running time of $\mathcal{A}$ is at most $t = N^{1/(2^k-1)}/(\alpha \log N)$, for some constant α to be fixed later. Such a value exists by the assumption on the node-averaged complexity of $\mathcal{A}$. Then, we update $w = \max\{w, n'\}$, where n' is the value in Lemma 39. Observe that by how the graphs are constructed (see Figure 9), $N = \Theta(\prod_{i=1}^{k} w_i) = \Theta(w^{\sum_{i=1}^{k} 2^{i-1}/(2^k-1)}) = \Theta(w)$, and hence $N \leq \gamma w$ for some constant $\gamma > 0$.

We define the function $f_{\Pi,k+1}$ on the input $P = (H, F_{\text{incoming}}, F_{\text{outgoing}}, \mathcal{L}_{\text{incoming}})$ as follows. First, construct the tree $T = \text{add}(2|\Sigma_{\text{out}}|N+1, \text{pump}(w, r(P)))$, and let H' be the pumped path corresponding to the nodes of H in T. Let v be the white node in the middle of the path H' (breaking ties arbitrarily), and let e be an arbitrary edge incident to v on H'. Let i be the layer number of v. We call this edge the *designated edge* of the path, and v the *designated node* of the path. The goal is to run $\mathcal{A}$ on T and use its output on e to define the function $f_{\Pi,k+1}$. Observe that the output of $\mathcal{A}$, and its runtime, depend on the random bits (and the random ID assignment) assigned to the nodes of T. Let B be the subset of random bits and ID assignments to the nodes of T satisfying that, if we execute $\mathcal{A}$ on v, its running time is strictly less than $w_i/(\beta \log n)$, for some constant β to be fixed later. If β is small enough, then the set B cannot be empty, since otherwise, by Lemma 39 and the Markov inequality, the node-averaged complexity would be too high. We consider all such assignments, and we take the output o on e that appears more often (breaking ties arbitrarily). Observe that this output appears with probability at least $1/|\Sigma_{\text{out}}|$.

Let $(H', F'_{\text{incoming}}, F'_{\text{outgoing}}, \mathcal{L}'_{\text{incoming}})$ be the result of applying Lemma 37 on P with parameter w_i. For each endpoint u of H', we can compute a label-set as follows. Let u_1 be the node, among the endpoints of the designated edge e, that is closest to u, and let $u_z = u$. Consider the subpath $u_1, \ldots, u_z$ of H' induced by u_1, u_z, and the nodes in between. We define the label-set of e as $\{o\}$. Then, we compute the label-set of each edge $e_i = \{u_i, u_{i+1}\}$, for $i = 1, \ldots, z$, by applying Definition 5 as if the current node u_i were to be a single rake node with outgoing edge e_i, where e_z is defined to be the edge outgoing from u. Observe that the label-sets L_1 and L_2 of the endpoints induce an independent class for H', because, for any choice $(l_1, l_2) \in L_1 \times L_2$, by construction, we can pick a feasible assignment for P. By Lemma 37, this independent class is valid also for H.

Function correctness. We need to show that the testing procedure succeeds when testing $f_{\Pi,k+1}$. Assume for a contradiction that the testing procedure fails, meaning that at some point an empty maximal class is obtained. Let T be the graph constructed by the testing procedure in which the empty class is obtained. Let $T' = \text{pump}(w, T)$. Observe that T' contains at least one temporary node (because, otherwise, T would not contain any compress path, meaning that the function $f_{\Pi,k+1}$ has not been used at all). Hence, we can construct $T'' = \text{add}(2|\Sigma_{\text{out}}|N+1, T')$. Let $\mathcal{L}$ be the partial labeling obtained by putting the most probable output (as previously defined) on each designated edge of T''. We show that, if we run $\mathcal{A}$ on T'', there is non-zero probability of obtaining a correct labeling for all nodes of the subtree T' that agrees with $\mathcal{L}$, implying a contradiction on the fact that the testing procedure fails.

Let S be the set of neighborhoods obtained by taking the radius-$w_i/(\beta \log n)$ neighborhood of each designated node of level i, for all i. We now use Lemma 39 to show that, with large enough probability, it holds that any node of the graph, within its running time, is not able to communicate with two nodes that are part of different neighborhoods of S.

Lemma 40. *We can choose α (in the assumption on the runtime of $\mathcal{A}$) so that, for any arbitrary constant c, with probability at least $1 - 1/n^c$, a node is able to communicate with the nodes of at most one neighborhood of S.*

Proof. We prove that, with high probability, a node at the beginning of a compress path P does not communicate with the neighborhood of S containing the designated node of P. This implies the claim, since a node can reach one neighborhood of higher (or same) layers, but for all the others it has to pass through one of the endpoints of the path containing the designated node of the neighborhood (see Figure 9).

Consider a compress path P of level i. This path is of length at least w_i. Let v be the designated node of the path. Consider one of the two subpaths P' of P induced by nodes at distance strictly larger than $w_i/(\beta \log N)$ from v. Observe that P' has length at least $w_i/2 - w_i/(2\beta \log N) - O(1) \geq w_i/4$. Hence, we can split P' into $\delta \log N$ disjoint subpaths of length at least $w_i/(8\delta \log N)$, for any chosen constant δ. For each subpath P_i, let v_i be the central node of the path, breaking ties arbitrarily.

By applying Lemma 39, we get that the expected runtime of each node v_i is at most $t = N^{2^{i-1}/(2^k-1)}/(\alpha' \log N)$, for some constant α' directly proportional to α. The subpaths are of length at least

$$w_i/(8\delta \log N) = w^{2^{i-1}/(2^k-1)}/(8\delta \log N) \geq (N/\gamma)^{2^{i-1}/(2^k-1)}/(8\delta \log N).$$

Hence, we can choose α small enough, as a function of δ and γ, so that the expected runtime of the nodes v_i is less than $1/4$ of the length of the subpaths. Observe that, by the Markov inequality, the probability that a node v_i runs for more than $1/2$ of the length of the subpaths is at most $1/2$, and also observe that these events are independent for each v_i. Finally, observe that, in order to be possible for a node at the beginning of the path P to communicate with a node in the neighborhood in S containing the designated node v, it must hold that all nodes v_i run for strictly more than $1/2$ of the length of the subpaths, and this happens with probability at most $1/2^{\delta \log N} = 1/N^{\delta}$. $\square$

Consider the following set of instances: start from T'' and, in each radius-$w_i/(\beta \log n)$ neighborhood of each designated node v of level i, for all i, assign to the nodes a random bit sequence (and a random ID assignment) satisfying that the runtime of v is bounded by $w_i/(\beta \log n)$. Observe that this assignment forces the designated nodes to output the previously computed most probable outputs. Complete the ID assignment randomly (by using IDs not already assigned and assigning a different ID to each node). While the assigned IDs may repeat in the graph (and in particular two different designated nodes may see the same ID within their runtime), we argue that it does not actually matter. In fact, we claim that, if we run $\mathcal{A}$ on these instances with modified random bits and IDs, the failure probability of $\mathcal{A}$ increases from p to at most $|\Sigma_{\text{out}}|p + 1/n^c$, for any chosen constant c. The reason is that, by Lemma 40, for each node v it holds with probability at least $1 - 1/n^c$ that, in its t-radius neighborhood, where t is the runtime of v, there is at most one neighborhood with modified random bits, and the modified assignment provided to the neighborhood could happen in T'' (in a real instance, with unmodified random bits, and a random valid ID assignment) with probability at least $1/|\Sigma_{\text{out}}|$. This implies that the failure probability of each node is bounded by $|\Sigma_{\text{out}}|/n + 1/n^c$.

Hence, by a union bound, no node of T' fails with probability at most $N(|\Sigma_{\text{out}}|/n + 1/n^c) \leq 2N|\Sigma_{\text{out}}|/n < 1$. Hence, by the probabilistic method, there exists an assignment of output labels that is valid for all the nodes of T'.

8 Open Questions

We conclude with some open questions. We showed that all problems that have $O(\log n)$ worst-case complexity can be solved with $O(\log^* n)$ node-averaged complexity. We leave open to determine for which problems this can be improved.

Open Problem 1. For which LCLs with $O(\log n)$ worst-case complexity can we obtain $o(\log^* n)$ node-averaged complexity?

We showed that all problems that have worst-case complexity $\Theta(n^{1/k})$ must have node-averaged complexity $\Omega(n^{1/(2^k-1)}/\log n)$. We conjecture that the $\log n$ factor is an artefact of our proof technique and that it should not be there.

Open Problem 2. Can we prove a lower bound of $\Omega(n^{1/(2^k-1)})$ rounds for the node-averaged complexity of all problems with worst-case complexity $\Theta(n^{1/k})$?

We showed that for some LCL problems that have worst-case complexity $\Theta(n^{1/k})$, we can provide an algorithm with node-averaged complexity $O(n^{1/(2^k-1)})$. It is not clear if this can be done for all problems with worst-case complexity $\Theta(n^{1/k})$.

Open Problem 3. Can we prove an upper bound of $O(n^{1/(2^k-1)})$ rounds for the node-averaged complexity of all problems that have worst-case complexity $\Theta(n^{1/k})$?

Open Problem 4. Can we prove a lower bound of $\Omega(n^{1/k})$ rounds for the node-averaged complexity of some problems that have worst-case complexity $\Theta(n^{1/k})$?

References

[1] Yehuda Afek, Shay Kutten, and Moti Yung. The local detection paradigm and its application to self-stabilization. *Theor. Comput. Sci.*, 186(1-2):199–229, 1997. doi:10.1016/S0304-3975(96)00286-1.

[2] Alkida Balliu, Sebastian Brandt, Yi-Jun Chang, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. The distributed complexity of locally checkable problems on paths is decidable. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 262–271. ACM Press, 2019. arXiv:1811.01672, doi:10.1145/3293611.3331606.

[3] Alkida Balliu, Sebastian Brandt, Yi-Jun Chang, Dennis Olivetti, Jan Studený, and Jukka Suomela. Efficient classification of locally checkable problems in regular trees. In *Proc. 36th International Symposium on Distributed Computing,(DISC 2022)*, pages 8:1–8:19, 2022. doi:10.4230/LIPIcs.DISC.2022.8.

[4] Alkida Balliu, Sebastian Brandt, Yuval Efron, Juho Hirvonen, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Classification of distributed binary labeling problems. In *Proc. 34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *LIPIcs*, pages 17:1–17:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.17.

[5] 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 *36th International Symposium on Distributed Computing, (DISC 2022)*, pages 9:1–9:21, 2022. doi:10.4230/LIPIcs.DISC.2022.9.

[6] Alkida Balliu, Sebastian Brandt, Dennis Olivetti, Jan Studený, Jukka Suomela, and Aleksandr Tereshchenko. Locally checkable problems in rooted trees. In *Proc. 40th ACM Symposium on Principles of Distributed Computing (PODC 2021)*, pages 263–272, 2021. doi:10.1145/3465084.3467934.

[7] Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. How much does randomness help with locally checkable problems? In *Proc. 39th ACM Symposium on Principles of Distributed Computing (PODC 2020)*, pages 299–308. ACM Press, 2020. arXiv:1902.06803, doi:10.1145/3382734.3405715.

[8] Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. Almost global problems in the LOCAL model. *Distributed Comput.*, 34(4):259–281, 2021. doi:10.1007/s00446-020-00375-2.

[9] Alkida Balliu, Keren Censor-Hillel, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Locally checkable labelings with small messages. In *35th International Symposium on Distributed Computing, DISC 2021*, pages 8:1–8:18, 2021. doi:10.4230/LIPIcs.DISC.2021.8.

[10] Alkida Balliu, Mohsen Ghaffari, Fabian Kuhn, and Dennis Olivetti. Node and edge averaged complexities of local graph problems. In *Proc. 41st ACM Symp. on Principles of Distributed Computing (PODC)*, pages 4–14, 2022.

[11] Alkida Balliu, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Dennis Olivetti, and Jukka Suomela. New classes of distributed time complexity. In *Proc. 50th ACM Symposium on Theory of Computing (STOC 2018)*, pages 1307–1318. ACM Press, 2018. arXiv:1711.01871, doi:10.1145/3188745.3188860.

[12] Alkida Balliu, Juho Hirvonen, Dennis Olivetti, and Jukka Suomela. Hardness of minimal symmetry breaking in distributed computing. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 369–378. ACM Press, 2019. arXiv:1811.01643, doi:10.1145/3293611.3331605.

[13] Leonid Barenboim, Michael Elkin, and Fabian Kuhn. Distributed $(\Delta + 1)$-coloring in linear (in Δ) time. *SIAM J. on Computing*, 43(1):72–95, 2015.

[14] Leonid Barenboim and Yaniv Tzur. Distributed symmetry-breaking with improved vertex-averaged complexity. In *Proc. 20th Int. Conf. on Distributed Computing and Networking (ICDCN)*, pages 31–40, 2019.

[15] Sebastian Brandt, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Patric R. J. Östergård, Christopher Purcell, Joel Rybicki, Jukka Suomela, and Przemyslaw Uznanski. LCL problems on grids. In *Proc. 36th ACM Symposium on Principles of Distributed Computing (PODC 2017)*, pages 101–110, 2017. doi:10.1145/3087801.3087833.

[16] Yi-Jun Chang. The complexity landscape of distributed locally checkable problems on trees. In *Proc. 34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *LIPIcs*, pages 18:1–18:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.18.

[17] Yi-Jun Chang, Qizheng He, Wenzheng Li, Seth Pettie, and Jara Uitto. Distributed edge coloring and a special case of the constructive lovász local lemma. *ACM Trans. Algorithms*, 16(1):8:1–8:51, 2020.

[18] 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. doi:10.1137/17M1117537.

[19] Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the LOCAL model. *SIAM J. Comput.*, 48(1):33–69, 2019. doi:10.1137/17M1157957.

[20] Yi-Jun Chang, Jan Studený, and Jukka Suomela. Distributed graph problems through an automata-theoretic lens. In *Proc. 28th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2021)*, LNCS. Springer, 2021. arXiv:2002.07659.

[21] Soumyottam Chatterjee, Robert Gmyr, and Gopal Pandurangan. Sleeping is efficient: MIS in $O(1)$-rounds node-averaged awake complexity. In *Proc. 39th ACM Symp. on on Principles of Distributed Computing (PODC)*, pages 99–108, 2020.

[22] Laurent Feuilloley. How long it takes for an ordinary node with an ordinary ID to output? In *Proc. 24th Int. Coll. on Structural Information and Communication Complexity (SIROCCO)*, volume 10641, pages 263–282, 2017.

[23] Manuela Fischer and Mohsen Ghaffari. Sublogarithmic distributed algorithms for Lovász local lemma, and the complexity hierarchy. In *Proc. 31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *LIPIcs*, pages 18:1–18:16, 2017. doi:10.4230/LIPIcs.DISC.2017.18.

[24] Christoph Grunau, Václav Rozhon, and Sebastian Brandt. The landscape of distributed complexities on trees and beyond. In *Proc. 41st ACM Symposium on Principles of Distributed Computing (PODC 2022)*, pages 37–47, 2022. doi:10.1145/3519270.3538452.

[25] Öjvind Johansson. Simple distributed *Delta*+1-coloring of graphs. *Inf. Process. Lett.*, 70(5):229–232, 1999.

[26] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *J. ACM*, 63(2):17:1–17:44, 2016.

[27] Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.

[28] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*, pages 478–489. IEEE Computer Society, 1985. URL: https://doi.org/10.1109/SFCS.1985.43, doi:10.1109/SFCS.1985.43.

[29] Moni Naor and Larry J. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

[30] Václav Rozhoň and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proc. 52nd ACM Symp. on Theory of Computing (STOC)*, pages 350–363, 2020.

A Additional Related Work About LCLs

LCLs were introduced by Naor and Stockmeyer [29], but locally checkable problems were studied in the distributed setting even before [1]. Since then, LCL problems have been studied a lot, and we now known, for different possible topologies of graphs, what kind of worst-case complexities are possible.

Paths and cycles. We know that in paths and cycles there are only three possible complexities: $O(1)$, $\Theta(\log^* n)$, $\Theta(n)$ [29, 18]. Moreover, we know that randomness does not help to solve problems faster.

Trees. The possible deterministic complexities in trees are $O(1)$, $\Theta(\log^* n)$, $\Theta(\log n)$, and $\Theta(n^{1/k})$ for all integer $k \geq 1$; randomness either helps exponentially or not at all, and only for problems with deterministic complexity $\Theta(\log n)$, that hence have randomized complexity either $\Theta(\log n)$ or $\Theta(\log \log n)$ [24, 12, 18, 19, 8, 16].

General graphs. In general graphs, some *complexity gaps* that hold in the case of trees still hold, but now there are also many dense areas; while in trees randomness either helps exponentially or not at all, there are cases on general graphs where randomness helps only polynomially [19, 18, 23, 30, 11, 7].

LCLs in other settings. On d-dimensional balanced toroidal grids, it is known that the only possible complexities are $O(1)$, $O(\log^* n)$, and $\Theta(n^{1/d})$ (even by allowing randomness) [15].

For problems that, in the black-white formalism (as defined in Section 3), can be expressed by using at most two labels, on regular trees the only possible deterministic complexities are $O(1)$, $\Theta(\log n)$ and $\Theta(n)$, implying that any LCL with complexity $\Theta(\log^* n)$ needs to be expressed with at least 3 labels [4].

It is known that any LCL on trees that can be solved in T rounds in the LOCAL model of distributed computing can be solved in $O(T)$ rounds in the CONGEST model; it is also known that this does not hold if we consider general graphs [9].

On *rooted* trees, it is known that all possible complexities are $O(1)$, $\Theta(\log^* n)$, $\Theta(\log n)$, $\Theta(n^{1/k})$ for all integer $k \geq 1$, but perhaps surprisingly, randomness never helps in solving problems faster [6].

LCLs have been studied also in other models of interest such as MPC [5].

Decidability. LCLs have been studied not only from a point of view of complexity theory. Researchers tried also to address the following questions. Given a specific LCL, can we decide its time complexity with a centralized algorithm? Is it possible to automate the design of algorithms for solving LCLs?

In general, the complexity of an LCL is not decidable: even on unlabeled non-toroidal grid graphs it is undecidable whether the complexity of an LCL is $O(1)$ [29]. However, there are some positive results in more restricted but still interesting settings.

On paths and cycles, it is possible to determine what is the time complexity of a given problem, but it becomes EXPTIME-hard if some input is provided to the nodes [29, 15, 20, 2, 16].

In unlabeled toroidal grids, it is decidable whether the complexity of an LCL is $O(1)$, but it is undecidable whether its complexity is $\Theta(\log^* n)$ or $\Theta(n)$ [15].

On trees, given an LCL, it is possible to decide on which side of the gap $\omega(\log n)$ – $n^{o(1)}$ its complexity lies. Moreover, it is decidable if an LCL has complexity $\Theta(n^{1/k})$ for some k, and it is also possible to determine the exact value of k [19, 16], but the algorithm is very far from being practical. However, if we restrict to regular trees with no inputs, then there is a practical polynomial-time algorithm (in the size of the description of the LCL problem) for deciding if an LCL has complexity $\Theta(n^{1/k})$ and determining the exact value of k [3]. Similarly, for rooted trees, there are efficient algorithms for determining the optimal asymptotic complexity of a given LCL [6, 3]. On unrooted regular trees with no input, if we restrict to problems that can be expressed by using at most two labels in the black-white formalism, the deterministic complexity of a problem is decidable [4].

It is still an open question whether, on trees, we can obtain decidability for the lower complexities, e.g., it is unknown whether we can decide if an LCL on trees can be solved in $O(1)$ rounds or if it requires $\Omega(\log^* n)$ rounds, and whether it can be solved in $O(\log^* n)$ rounds or if it requires $\Omega(\log n)$ with deterministic algorithms.

B An Algorithm For Solving All LCLs in $O(D)$ Rounds

In order to give more intuition about the generic method that can be used to solve LCL problems, we repropose a simplified setting already presented in [9], where we restrict a bit the class of problems that we consider, as follows. We are given a tree of constant maximum degree, where to each edge is assigned an input label that comes from a finite set, and the goal is to assign an output label to each edge, also from a finite set, in such a way that, for each node, the multiset of incident input-output pairs of labels is contained in a list of given allowed configurations (this list is the same for all nodes). In other words, in this simplified setting, an LCL problem is described by providing a list of allowed configurations C, that are multisets of input-output pairs of labels. We now describe a procedure, already presented in [9], that solves any problem of this form in $O(D)$ rounds, where D is the diameter of the tree. The procedure works as follows:

1. Each leaf node v, as a function of the input label ℓ_i of its incident edge e and the list of allowed configurations C, computes the set S_v of output labels ℓ_o satisfying that the multiset $\{(\ell_i, \ell_o)\}$ is in C, that is, v computes the set S_v of output labels ℓ_o that, if assigned to e, would make v happy.

2. Each leaf v sends the set S_v to its neighbor.

3. Leaves are removed from the tree. Let L be the set of nodes that became leaves after the removal operation. Each node $v \in L$ proceeds as follows. Let $u_1, \ldots, u_d$ be the neighbors of v that got removed in previous steps, let $\ell_{i_1}, \ldots, \ell_{i_d}$ be the input labels on the edges connecting node v to the nodes $u_1, \ldots, u_d$, and let ℓ_i be the input label connecting v to its neighbor that is still present. Node v computes the set S_v of output labels ℓ_o satisfying that there exists a choice $(\ell_{o_1}, \ldots, \ell_{o_d}) \in S_{u_1} \times \ldots \times S_{u_d}$ such that the multiset $\{(\ell_i, \ell_o), (\ell_{i_1}, \ell_{o_1}), \ldots, (\ell_{i_d}, \ell_{o_d})\}$ is in C. In other words, node v computes the set S_v of output labels for its remaining edge satisfying that, even if the output from S_v is chosen adversarially, there is still a choice that v can make, over the sets received from its removed neighbors, that would give an assignment of labels that makes v happy. Then, node v sends S_v to its remaining neighbor.

4. Repeat step 3 until the graph is empty.

5. The last removed node chooses a label from each received set, in such a way that the resulting multiset of input-output pairs of labels is a configuration in C. (It may happen that two neighboring nodes get removed last, at the same time, but this case can be handled in a similar way.)

6. Removed nodes are put back in reverse order. Observe that, when a node v is put back, the output label ℓ_o of the edge connecting v to its only neighbor that is currently present has already been assigned, and that this output label ℓ_o is in S_v. Node v picks a label from the sets assigned to the edges connecting v to its other neighbors (that is, neighbors that are going to be put back in the next step) in a way that is compatible with ℓ_o.

It is not difficult to see that this algorithm computes a correct solution for the problem, assuming that the computed sets never become empty, that is, as long as, for each $v \in V$, $S_v \neq \emptyset$. Also, it is clear that the time complexity of this algorithm is $O(D)$. It has been shown in [9] that, if it happens that some node v gets $S_v = \emptyset$, then it means that the problem is unsolvable. Moreover, [9] showed that by following this algorithm, we get a generic way to solve all problems in $O(D)$ rounds that is actually bandwidth efficient, since the sets that are sent at each step have constant size.

Improving the round complexity. The previously described procedure shows how to solve all (solvable) problems in $O(D)$ rounds, which is nice for the CONGEST model, but is a trivial result in the LOCAL model. Moreover, the diameter D could be as high as $\Omega(n)$, and for some problems this complexity may be suboptimal. Hence, in some cases we would like to obtain a faster algorithm, and this is what has been shown in [19, 16]. The idea of [19, 16] is that, if, instead of removing only nodes of degree 1 at each step, we also sometimes remove nodes of degree 2 as well, then it takes less steps to obtain an empty graph. One of the issues to handle when running this modified procedure is that we now have to find a way to assign sets to nodes of degree 2, which may form long paths, and this is the part that turns out to be quite challenging.

Restricting to a specific kind of LCLs. In order to prove our results, we use and extend ideas presented in [19, 16, 9]. In particular, in [9] it has been shown that, in bounded-degree trees, if an LCL problem has time complexity T in the LOCAL model, then it has time complexity $O(T)$ in the CONGEST model, both for deterministic and randomized worst-case complexities. In order to show this result, the authors of [9] not only extended the results of [19, 16], but also provided a more accessible version of some of the proofs of [19, 16]. The reason why these proofs are more accessible is that [9] does not show results for standard LCLs, but only for a restriction of those, that are LCLs that can be expressed in a formalism called black-white. Importantly, the authors also showed that, if we restrict to trees, for any standard LCL (LCLs as they are usually defined in the literature), we can define an LCL in the black-white formalism that has the same complexity of the original one, up to an *additive* constant. In other words, considering LCLs in the black-white formalism is not really a restriction if the graph class we are working on is trees (in general graphs this turns out to not be the case). In this paper, we follow a similar route of [9]: in order to keep our proofs more accessible we prove our statements for LCLs expressed in the black-white formalism, but we also prove that, for any standard LCL, we can define an LCL in the black-white formalism that has the same node-averaged complexity of the original one, up to a *multiplicative* constant factor, implying that our results hold for all LCLs as well. This equivalence is shown in Lemma 45.

C Different Ways to Define LCLs

In this section we prove an equivalence, for node-averaged complexity, between different definitions of LCLs. We start by providing the standard definition of LCLs, that in the following we will call standard LCLs.

Locally Checkable Labeling problems. An LCL problem Π is defined as a tuple $(\Sigma_{\text{in}}, \Sigma_{\text{out}}, C, r)$, where:

- Σ_{in} and Σ_{out} are finite sets of labels that represent the possible input and output labels.
- The parameter $r \geq 1$ is an integer, and it represents the so-called *checkability radius* of the LCL problem Π.
- C is a finite set of labeled graphs that represent allowed neighborhoods, and more precisely C is a finite set of pairs (H, v), where:
  - $H = (V_H, E_H)$ is a graph, and $v \in V_H$.
  - The eccentricity of v in H is at most r.
  - To each pair $(v, e) \in V_H \times E_H$ is assigned a label $\ell_{\text{in}} \in \Sigma_{\text{in}}$ and a label $\ell_{\text{out}} \in \Sigma_{\text{out}}$.

Solving an LCL problem Π on a graph $G = (V, E)$ means that:

- To each node-edge pair $(v, e) \in V \times E$ is assigned a label $i(e) \in \Sigma_{\text{in}}$.
- The task is to assign a label $o(e) \in \Sigma_{\text{out}}$ to each node-edge pair $(v, e) \in V \times E$ such that, for each node $v \in V$, it holds that the (labeled) graph N_v^r induced by v's radius-r neighborhood is in C, that is, N_v^r is isomorphic to a (labeled) graph contained in C.

Node-edge formalism. Using the black-white formalism (as defined in Section 3) requires the graph to be properly 2-colored. We now describe a formalism, called node-edge, that does not have this requirement. We will soon show an equivalence between the node-edge formalism and the black-white formalism. A problem Π described in the node-edge formalism is a tuple $(\Sigma_{\text{in}}, \Sigma_{\text{out}}, C_N, C_E)$, where:

- Σ_{in} and Σ_{out} are finite sets of labels.
- C_N and C_E are both multisets of pairs, where each pair $(\ell_{\text{in}}, \ell_{\text{out}})$ is in $\Sigma_{\text{in}} \times \Sigma_{\text{out}}$. The multisets in C_E have size exactly 2.

Solving a problem Π on a graph G means that:

- To each node-edge pair $(v, e) \in V \times E$ is assigned a label $i(e) \in \Sigma_{\text{in}}$.
- The task is to assign a label $o(e) \in \Sigma_{\text{out}}$ to each node-edge pair $(v, e) \in V \times E$ such that, for each node $v \in V$ (resp. each edge $e \in E$) it holds that the multiset of incident input-output pairs is in C_V (resp. in C_E).

Equivalence between black-white and node-edge formalisms. Observe that, if we focus only on what is a problem, and we forget about what it means to solve a problem, it is clear that the node-edge formalism is a subcase of the more general black-white formalism. In fact, it is easy to see that a problem Π in the node-edge formalism implicitly defines also a problem Π' in the black-white formalism, while a problem Π' in the black-white formalism implicitly defines also a problem Π in the node-edge formalism if it satisfies that C_B contains only multisets of size exactly 2.

Moreover, it has been show in [9] that also the time complexities of Π and Π' are related. In fact, one can prove that, given an algorithm A for Π, we can define a new algorithm A' for Π' with the same asymptotic worst-case round complexity, and vice versa.

We now describe the ideas that show that, given an algorithm A' for Π', we can use it to solve Π (the other direction can be shown in a similar way). The idea is to simulate the algorithm for Π' on a virtual graph G', defined as a function of the real graph $G = (V, E)$ as follows. Let $G' = (W \cup B, E')$, where $W = V$, $B = E$, and we connect $v \in V$ with $e \in B$ if $v \in e$. In other words, original nodes are white, and we add a black node in the middle of each original edge. Observe that, by solving Π' on G', and then mapping the output obtained for the edges of G' to the node-edge pairs of G, we also solve Π on G. Also, observe that simulating the execution of a T round algorithm for G' only costs $T/2 + O(1)$ rounds on G.

Equivalence between standard LCLs and LCLs in the node-edge formalism. In [9] it has been shown that, given a standard LCL Π on trees, it is possible to define a node-edge checkable problem Π' satisfying that, given a solution for Π, it is possible to spend $O(1)$ rounds to solve Π', and vice versa.

Lemma 41 ([9]). *For any LCL problem Π on trees with checkability radius $r = O(1)$ we can define a node-edge checkable problem Π' that satisfies the following:*

- *There exists an $O(1)$-rounds algorithm that, given in input a solution for Π', outputs a solution for Π.*
- *There exists an $O(1)$-rounds algorithm that, given in input a solution for Π, outputs a solution for Π'.*

By combining this lemma with the equivalence that we discussed before, we obtain that by studying the worst-case complexity of LCL problems on trees in the black-white formalism is not a restriction, and hence we get results that hold for LCLs in the more general form. Figure 10 shows an example of an LCL problem defined in different formalisms. In the following we show a similar statement for the node-averaged complexity of LCLs.

Equivalence between standard LCLs and black-white formalism for node-averaged complexity. In order to prove that we can restrict to the black-white formalism also for the case of node-averaged complexity, we first prove an equivalence between standard LCLs and the node-edge formalism, and then we prove an equivalence between the node-edge formalism and the black-white formalism.

In the case of worst case complexity, if we have an algorithm A_1 that produce a result in T_1 rounds, and we have a different algorithm A_2 that takes the output of A_1 in input and produces a result in T_2 rounds, then we can execute A_2 after A_1 and obtain a running time of $T_1 + T_2$. For node-averaged complexity, unfortunately, the same does not hold. Nevertheless we can achieve something similar.

Lemma 42. *Suppose that we have an algorithm A_1 with node-averaged complexity T_1, and an algorithm A_2 with worst-case complexity T_2. Consider the algorithm A_3 obtained by combining A_1 and A_2, where nodes execute A_2 only after all nodes in their T_2 radius neighborhood terminated the execution of A_1. The node-averaged complexity of A_3 is $O(T_1 \cdot \Delta^{T_2})$.*

Proof. We show a charging scheme that satisfies that the sum of the charges is an upper bound on the sum of the running times of the nodes. For each node v of the graph, let $t(v)$ be the last node in the T_2-radius neighborhood of v that terminates the execution of A_1, break ties arbitrarily. We charge the running time of v in A_3 to $t(v)$. Observe that each node is charged $O(\Delta^{T_2})$ times, and that the sum of the running times in A_1 of the nodes charged at least once is $O(n \cdot T_1)$. Hence, the sum of the charges, and the running times in A_3, is $O(n \cdot T_1 \cdot \Delta^{T_2})$, implying that the node-averaged complexity of A_3 is $O(T_1 \cdot \Delta^{T_2})$, as required. □

We are now ready to prove an equivalence, for node-averaged complexity, between standard LCLs and LCLs in the node-edge formalism.

Lemma 43. *For any LCL problem Π on trees with checkability radius r and node-averaged complexity T we can define a node-edge checkable problem Π' with node-averaged complexity $\Theta(T)$.*

Proof. We apply Lemma 42 where T_1 is the node-averaged complexity of the given LCL problem Π, and T_2 is the worst-case time complexity required to convert a solution for Π into a solution of its equivalent node-edge checkable variant. By Lemma 41 we have that $T_2 = O(1)$. Since $\Delta = O(1)$, then the claim follows. □

We now prove an equivalence, for node-averaged complexity, between LCLs in the node-edge formalism and LCLs in the black-white formalism.

Lemma 44. *For any node-edge checkable LCL problem Π on trees with node-averaged complexity T we can define an LCL Π' in the black-white formalism with node-averaged complexity $\Theta(T)$.*

Proof. Let $\Pi = (\Sigma_{\text{in}}, \Sigma_{\text{out}}, C_N, C_E)$. We define $\Pi' = \Pi$, that is, the input and output labels are exactly the same, the white constraint of Π' is C_N, and the black constraint of Π' is C_E. Observe that, while Π and Π' are syntactically the same, an algorithm for Π is designed for working on a tree with no 2-coloring given, and it needs to assign an output label to each node-edge pair, while

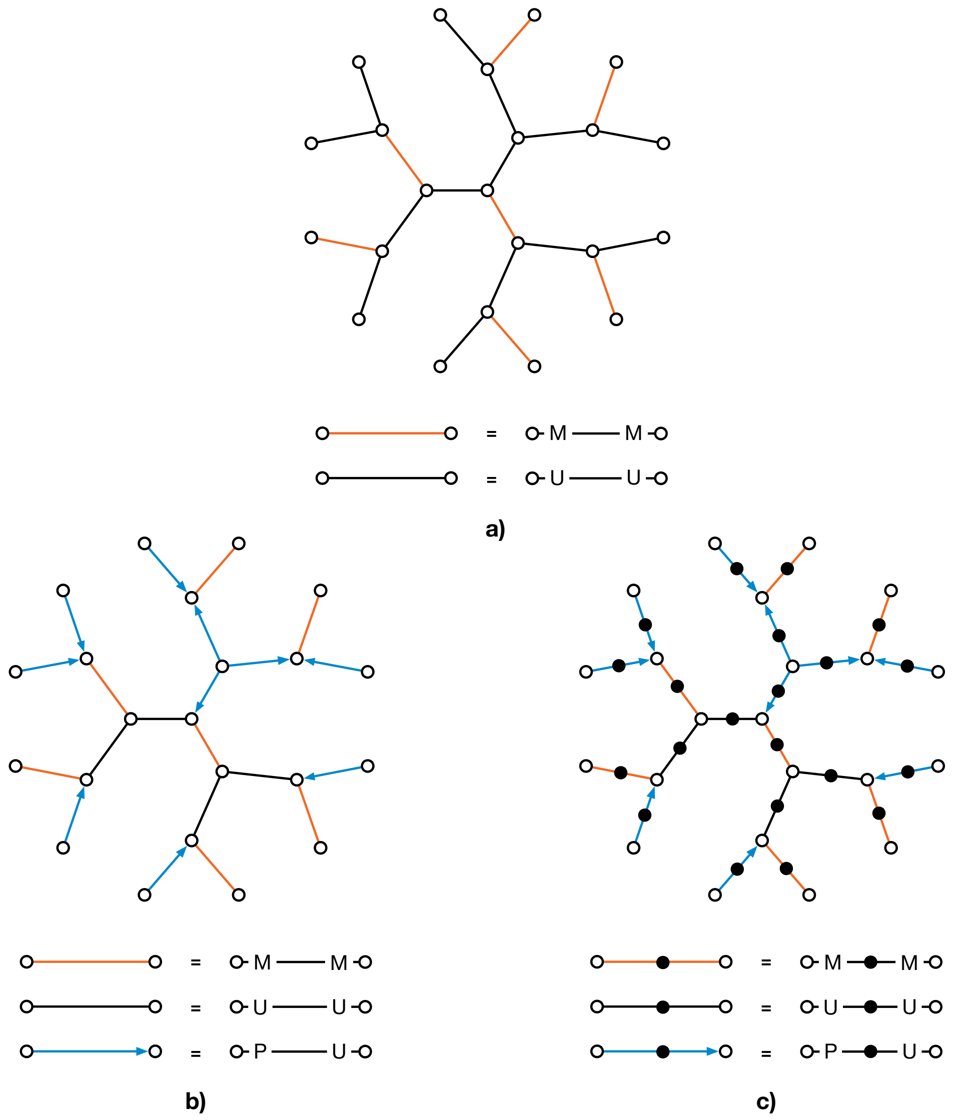

Figure 10: a) In the maximal matching problem, a node is either matched or all its neighbors are matched. This can be described as an LCL by letting each edge be either labeled $\{M, M\}$ or $\{U, U\}$, and then listing all possible valid radius-2 neighborhoods. b) The maximal matching problem encoded in the node-edge formalism. One way is to require unmatched nodes to orient all their edges outgoing, and then require all nodes with at least one incoming edge to be matched. These constraints can be given in the node-edge formalism. c) The 2-colored graph in which the maximal matching problem has the same complexity as in the node-edge setting.

an algorithm for Π' is designed for working on a tree that is properly 2-colored and where all black nodes have degree 2, and it needs to assign an output label to each edge.

We first show that, given an algorithm A for solving Π with node-averaged complexity T, we can design an algorithm A' for solving Π' with node-averaged complexity $O(T)$. The algorithm A' works as follows. White nodes simulate the execution of A, while black nodes relay messages that are exchanged between white nodes. This algorithm clearly solves Π'. Each white node spends exactly twice the running time of A, since each round of A is simulated with 2 rounds of A' (it takes 2 rounds for white nodes to exchange messages). Observe that black nodes, in order to know their output, need to wait that the two incident white neighbors terminate. Hence, we can charge their running time to their slowest neighbor. Since each node is charged at most $\Delta = O(1)$ times, then we obtain that A' has node-averaged complexity $O(T)$.

We now show that, given an algorithm A' for solving Π' with node-averaged complexity T, we can design an algorithm A for solving Π with node-averaged complexity $O(T)$. The algorithm A is defined as follows. Each edge of the tree is assigned to one incident node, arbitrarily. In the algorithm A, the nodes pretend to be white nodes, and simulate the execution of A'. Moreover, the nodes simulate, for each incident edge assigned to them, the execution of a black node, that is connected to the two nodes incident to the edge. This algorithm clearly solves Π, and its running time can be bounded as follows. The running time of each node is charged to its longest running simulated (black or white) node. Observe that each simulated node is charged at most once, and that the sum of the running times of the simulated nodes that have been charged at least once is an upper bound on the sum of the running times in A. This amount is at most $n \cdot T$, where n is the size of the simulated graph. Since the number of nodes in the real graph and in the simulated one is asymptotically the same, then we obtain that the node-averaged complexity of A is at most $O(T)$. □

By combining Lemma 43 and Lemma 44 we obtain that focusing on LCLs in the black-white formalism is not a restriction, and that our results apply to standard LCLs as well. In particular, we obtain the following.

Lemma 45. *For any LCL problem Π on trees with checkability radius r and node-averaged complexity T we can define an LCL Π' in the black-white formalism with node-averaged complexity $\Theta(T)$.*