Almost global problems in the LOCAL model

Alkida Balliu[1] · Sebastian Brandt[2] · Dennis Olivetti[1] · Jukka Suomela[1]

Abstract

The landscape of the distributed time complexity is nowadays well-understood for subpolynomial complexities. When we look at deterministic algorithms in the LOCAL model and locally checkable problems (LCLs) in bounded-degree graphs, the following picture emerges:

– There are lots of problems with time complexities of $\Theta(\log^* n)$ or $\Theta(\log n)$.
– It is not possible to have a problem with complexity between $\omega(\log^* n)$ and $o(\log n)$.
– In *general graphs*, we can construct LCL problems with infinitely many complexities between $\omega(\log n)$ and $n^{o(1)}$.
– In *trees*, problems with such complexities do not exist.

However, the high end of the complexity spectrum was left open by prior work. In general graphs there are LCL problems with complexities of the form $\Theta(n^\alpha)$ for any rational $0 < \alpha \le 1/2$, while for trees only complexities of the form $\Theta(n^{1/k})$ are known. No LCL problem with complexity between $\omega(\sqrt{n})$ and $o(n)$ is known, and neither are there results that would show that such problems do not exist. We show that:

– In *general graphs*, we can construct LCL problems with infinitely many complexities between $\omega(\sqrt{n})$ and $o(n)$.
– In *trees*, problems with such complexities do not exist.

Put otherwise, we show that any LCL with a complexity $o(n)$ can be solved in time $O(\sqrt{n})$ in trees, while the same is not true in general graphs.

1 Introduction

Recently, in the study of *distributed graph algorithms*, there has been a lot of interest on *structural complexity theory*: instead of studying the distributed time complexity of specific graph problems, researchers have started to put more focus on the study of *complexity classes* in this context.

1.1 LCL problems

A particularly fruitful research direction has been the study of distributed time complexity classes of so-called LCL problems (locally checkable labellings). We will define LCLs formally in Sect. 2.2, but the informal idea is that LCLs are graph problems in which *feasible solutions can be verified by checking all constant-radius neighbourhoods*. Examples

✉ Jukka Suomela
jukka.suomela@aalto.fi

Alkida Balliu
alkida.balliu@aalto.fi

Sebastian Brandt
brandts@ethz.ch

Dennis Olivetti
dennis.olivetti@aalto.fi

[1] Aalto University, Espoo, Finland

[2] ETH Zürich, Zurich, Switzerland

of such problems include vertex colouring with k colours, edge colouring with k colours, maximal independent sets, maximal matchings, and sinkless orientations.

LCLs play a role similar to the class NP in the centralised complexity theory: these are problems that would be easy to solve with a *nondeterministic* distributed algorithm—guess a solution and verify it in $O(1)$ rounds—but it is not at all obvious what is the distributed time complexity of solving a given LCL problem with *deterministic* distributed algorithms.

1.2 Distributed structural complexity

In the classical (centralised, sequential) complexity theory one of the cornerstones is the *time hierarchy theorem* [15]. In essence, it is known that giving more time always makes it possible to solve more problems. Distributed structural complexity is fundamentally different: there are various *gap results* that establish that there are no LCL problems with complexities in a certain range. For example, it is known that there is no LCL problem whose deterministic time complexity on bounded-degree graphs is between $\omega(\log^* n)$ and $o(\log n)$ [8].

Such gap results have also direct applications: we can *speed up* algorithms for which the current upper bound falls in one of the gaps. For example, it is known that Δ-colouring in bounded-degree graphs can be solved in polylog n time [20]. Hence 4-colouring in 2-dimensional grids can be also solved in polylog n time. But we also know that in 2-dimensional grids there is a gap in distributed time complexities between $\omega(\log^* n)$ and $o(\sqrt{n})$ [6], and therefore we know we can solve 4-colouring in $O(\log^* n)$ time.

The ultimate goal here is to identify all such gaps in the landscape of distributed time complexity, for each graph class of interest.

1.3 State of the art

Some of the most interesting open problems at the moment are related to *polynomial complexities in trees*. The key results from prior work are:

– In bounded-degree trees, for each positive integer k there is an LCL problem with time complexity $\Theta(n^{1/k})$ [9].
– In bounded-degree graphs, for each rational number $0 < \alpha \le 1/2$ there is an LCL problem with time complexity $\Theta(n^\alpha)$ [2].

However, there is no separation between trees and general graphs in the polynomial region. Furthermore, we do not have any LCL problems with time complexities $\Theta(n^\alpha)$ for any $1/2 < \alpha < 1$.

1.4 Our contributions

This work resolves both of the above questions. We show that:

– In bounded-degree graphs, for each rational number $1/2 < \alpha < 1$ there is an LCL problem with time complexity $\Theta(n^\alpha)$.
– In bounded-degree trees, there is no LCL problem with time complexity between $\omega(\sqrt{n})$ and $o(n)$.

Hence whenever we have a slightly sublinear algorithm, we can always speed it up to $O(\sqrt{n})$ in trees, but this is not always possible in general graphs.

1.5 Key techniques

We use ideas from the classical centralised complexity theory—e.g. Turing machines and regular languages—to prove results in distributed complexity theory.

The key idea for showing that there are LCLs with complexities $\Theta(n^\alpha)$ in bounded-degree graphs is that we can take any *linear bounded automaton M* (a Turing machine with a bounded tape), and construct an LCL problem Π_M such that the *distributed* time complexity of Π is a function of the *sequential* running time of M. Prior work [2] used a class of *counter machines* for a somewhat similar purpose, but the construction in the present work is much simpler, and Turing machines are more convenient to program than the counter machines used in the prior work.

To prove the gap result, we heavily rely on Chang and Pettie's [9] ideas: they show that one can relate LCL problems in trees to regular languages and this way generate equivalent subtrees by "pumping". However, there is one fundamental difference:

– Chang and Pettie first construct certain *universal* collections of tree fragments (that do not depend on the input graph), use the existence of a fast algorithm to show that these fragments can be labelled in a convenient way, and finally use such a labelling to solve any given input efficiently.
– We work directly with the *specific* input graph, expand it by "pumping", and apply a fast algorithm there directly.

Many speedup results make use of the following idea: given a graph with n nodes, we pick a much *smaller* value $n' \ll n$ and lie to the algorithm that we have a tiny graph with only n' nodes [6,8]. Our approach essentially reverses this: given a graph with n nodes and an algorithm $\mathcal{A}$, we pick a much *larger* value $n' \gg n$ and lie to the algorithm that we have a huge graph with n' nodes.

1.6 Open problems

Our work establishes a gap between $\Theta(n^{1/2})$ and $\Theta(n)$ in trees. The next natural step would be to generalise the result and establish a gap between $\Theta(n^{1/(k+1)})$ and $\Theta(n^{1/k})$ for all positive integers k.

2 Model and related work

As we study LCL problems, a family of problems defined on *bounded-degree graphs*, we assume that our input graphs are of degree at most Δ, where $\Delta = O(1)$ is a known constant. Each input graph $G = (V, E)$ is simple, connected, and undirected; here V is the set of nodes and E is the set of edges, and we denote by $n = |V|$ the total number of nodes in the input graph.

2.1 Model of computation

The model considered in this paper is the well studied LOCAL model [17,21]. In the LOCAL model, each node $v \in V$ of the input graph G runs the same deterministic algorithm. The nodes are labelled with unique $O(\log n)$-bit identifiers, and initially each node knows only its own identifier, its own degree, and the total number of nodes n.

Computation proceeds in synchronous rounds. At each round, each node

– sends a message to its neighbours (it may be a different message for different neighbours),
– receives messages from its neighbours,
– performs some computation based on the received messages.

In the LOCAL model, there is no restriction on the size of the messages or on the computational power of a node. The time complexity of an algorithm is measured as the number of communication rounds that are required such that every node is able to stop and produce its local output. Hence, after t rounds in the LOCAL model, each node can gather the information in the network up to distance t from it. In other words, in t rounds a node can gather all information within its t-radius neighbourhood, where the t-radius neighbourhood of a node v is the subgraph containing all nodes at distance at most t from v and all edges incident to nodes at distance at most $t-1$ from v (including the inputs given to these nodes). Also, in t rounds, the information outside the t-radius neighbourhood of a node v cannot reach v. This means that a t-round algorithm running in the LOCAL model can be seen as a function that maps all possible t-radius neighbourhoods to the outputs. Notice that, in the LOCAL model, every problem can be solved in *diameter* number of rounds, where the diameter of a graph G is defined as the largest hop-distance among any pair of nodes in G. In fact, in diameter time each node can gather all information there is in the whole graph and solve the problem locally.

2.2 Locally checkable labellings

Locally checkable labelling problems (LCLs) were introduced in the seminal work of Naor and Stockmeyer [18]. Informally, LCLs are graph problems defined on bounded-degree graphs (i.e., graphs where the maximum degree is constant with respect to the number of nodes), where nodes have as input a label from a constant-size set of input labels, and they must produce an output from a constant-size set of output labels. The validity of these output labels is determined by a set of local constraints.
Formal definition. Let $\mathcal{F}$ be the family of bounded-degree graphs. An LCL is defined as a tuple $\Pi = (\Sigma_{\mathsf{in}}, \Sigma_{\mathsf{out}}, C, r)$ as follows.

– Σ_{in} and Σ_{out} are constant-size sets of labels;
– r is an arbitrary constant (called the checkability radius of the problem);
– C is a set of graphs where

  – each graph $H \in C$ is centred at some node v,
  – the distance of v from all other nodes in H, i.e., the radius of v, is at most r,
  – each node u is labelled with a pair $(i(u), o(u)) \in \Sigma_{\mathsf{in}} \times \Sigma_{\mathsf{out}}$.

An example. An example of an LCL problem is vertex 3-colouring, where $\Sigma_{\mathsf{in}} = \{\bot\}$, $\Sigma_{\mathsf{out}} = \{1, 2, 3\}$, $r = 1$, and C is defined as all graphs of radius 1 in $\mathcal{F}$ such that each node has a colour in Σ_{out} that is different from the ones of its neighbours.
Solving a problem. In general, solving an LCL means the following. We are given a graph $G = (V, E) \in \mathcal{F}$ and an input assignment $i\colon V \to \Sigma_{\mathsf{in}}$. The goal is to produce an output assignment $o\colon V \to \Sigma_{\mathsf{out}}$. Let $B(v)$ be the subgraph of G induced by nodes of distance at most r from v, augmented with the inputs and outputs assigned by i and o. The output assignment is valid if and only if, for each node $v \in V$, we have $B(v) \in C$. In that case, we call (G, i, o) a valid configuration.

This can be adapted to a distributed setting in a straightforward manner: if we are solving an LCL in the LOCAL model with a distributed algorithm $\mathcal{A}$, the input graph $G = (V, E) \in \mathcal{F}$ is the communication network, each node v initially knows only its own part of the input $i(v) \in \Sigma_{\mathsf{in}}$, and when algorithm $\mathcal{A}$ stops, each node v has to know its own part of output $o(v) \in \Sigma_{\mathsf{out}}$. The local outputs have to form a valid configuration (G, i, o).

Distributed time complexity. The distributed time complexity of an LCL problem Π in a graph family $\mathcal{F}$ is the pointwise smallest $t : \mathbb{N} \to \mathbb{N}$ such that there is a distributed algorithm $\mathcal{A}$ that solves Π in $t(n)$ communication rounds in any graph $G \in \mathcal{F}$ with n nodes, for any $n \in \mathbb{N}$, and for any input labelling of G.

Distributed verifiers. Above, we have defined an LCL as a set C of correctly labelled subgraphs. Equivalently, we could define an LCL in terms of a *verifier* $\mathcal{A}'$. A verifier is a distributed algorithm that receives both i and o as inputs, runs for r communication rounds, and then each node v outputs either 'accept' or 'reject'. We require that the output of $\mathcal{A}'$ does not depend on the ID assignment or on the size of the input graph, but only on the structure of G and input and output labels in the r-radius neighbourhood of v. Now we say that (G, i, o) is a valid configuration if all nodes output 'accept'.

This is equivalent to the above definition, as in r communication rounds each node v can gather all information within distance r, and nothing else. Hence $\mathcal{A}'$ can output 'accept' if $B(v) \in C$; equivalently, the output of any such algorithm $\mathcal{A}'$ defines a set C of correctly labelled neighbourhoods.

If $\mathcal{A}$ solves an LCL problem Π in time $t(n)$, and $\mathcal{A}'$ is the verifier for Π, then by definition the composition of $\mathcal{A}$ and $\mathcal{A}'$ is a distributed algorithm that runs in $t(n) + r$ rounds and always outputs 'accept' everywhere. It is important to note that, while the output of algorithm $\mathcal{A}$ may depend on the ID assignment that nodes have, the output of verifier $\mathcal{A}'$ must not depend on the ID assignment or on the size of the graph.

2.3 Related work

Cycles and paths. LCL problems are fully understood in the case of cycles and paths. In these graphs it is known that there are LCL problems having complexities $O(1)$, e.g. trivial problems, $\Theta(\log^* n)$, e.g. 3-vertex colouring, and $\Theta(n)$, e.g. 2-vertex colouring [10,17]. Chang et al. [8] showed two automatic speedup results: any $o(\log^* n)$-time algorithm can be converted into an $O(1)$-time algorithm; any $o(n)$-time algorithm can be converted into an $O(\log^* n)$-time algorithm. They also showed that randomness does not help in comparison with deterministic algorithms in cycles and paths.

Oriented grids. Brandt et al. [6] studied LCL problems on oriented grids, showing that, as in the case of cycles and paths, the only possible complexities of LCLs are $O(1)$, $\Theta(\log^* n)$, and $\Theta(n)$, on $n \times n$ grids, and it is also known that randomness does not help [8,12]. However, while it is decidable whether a given LCL on cycles can be solved in t rounds in the LOCAL model [6,18], it is not the case for oriented grids [6].

Trees. Although well studied, LCLs on trees are not fully understood yet. Chang and Pettie [9] show that any $n^{o(1)}$-time algorithm can be converted into an $O(\log n)$-time algorithm. In the same paper they show how to obtain LCL problems on trees having deterministic and randomized complexity of $\Theta(n^{1/k})$, for any integer k. However, it is not known if there are problems of complexities between $o(n^{1/k})$ and $\omega(n^{1/(k+1)})$. Regarding decidability on trees, given an LCL it is possible to decide whether it has complexity $O(\log n)$ or $n^{O(1)}$ [9]. In other words, it is possible to decide on which side of the gap between $\omega(\log n)$ and $n^{o(1)}$ an LCL lies, but it is still an open question if we can decide whether a given LCL has complexity $O(\log^* n)$ or $\Omega(\log n)$.

General graphs. Another key direction of research is understanding LCLs on general (bounded-degree) graphs. Using the techniques presented by Naor and Stockmeyer [18], it is possible to show that any $o(\log \log^* n)$-time algorithm can be sped up to $O(1)$ rounds. It is known that there are LCL problems with complexities $\Theta(\log^* n)$ [3,4,11,19] and $\Theta(\log n)$ [5,8,13]. On the other hand, Chang et al. [8] showed that there are no LCL problems with deterministic complexities between $\omega(\log^* n)$ and $o(\log n)$. It is known that there are problems (for example, Δ-colouring) that require $\Omega(\log n)$ rounds [5,7], for which there are algorithms solving them in $O(\text{polylog } n)$ rounds [20]. Until very recently, it was thought that there would be many other gaps in the landscape of complexities of LCL problems in general graphs. Unfortunately, it has been shown in [2] that this is not the case: it is possible to obtain LCLs with numerous different deterministic time complexities, including $\Theta(\log^\alpha n)$ and $\Theta(\log^\alpha \log^* n)$ for any $\alpha \geq 1$, $2^{\Theta(\log^\alpha n)}$, $2^{\Theta(\log^\alpha \log^* n)}$, and $\Theta((\log^* n)^\alpha)$ for any $\alpha \leq 1$, and $\Theta(n^\alpha)$ for any $\alpha < 1/2$ (where α is a positive rational number).

3 Near-linear complexities in general graphs

In this section we show that there are LCLs with time complexities in the spectrum between $\omega(\sqrt{n})$ and $o(n)$. To show this result, we prove that we can take any linear bounded automaton (LBA) M, that is, a Turing machine with a tape of a bounded size, and an integer $i \geq 2$, and construct an LCL problem Π_M^i, such that the distributed time complexity of Π_M^i is related to the choice of i and to the sequential running time of M when starting from an empty tape.

In particular, given an LBA M, we will define a family of graphs, that we call *valid instances*, where nodes are required to output an encoding of the execution of M. An LCL must be defined on any (bounded-degree) graph, without any promise on the graph structure, thus, we will define the LCL Π_M^i by requiring nodes to either prove that the given instance is not a valid instance, or to output a correct encoding of the execution of M if the instance is a valid one. The manner in which the execution has to be encoded ensures that the complexity of the LCL Π_M^i will depend on the running time of the LBA M, and by constructing LBAs with suitable running times, we can show our result. The key idea here is that we will use valid
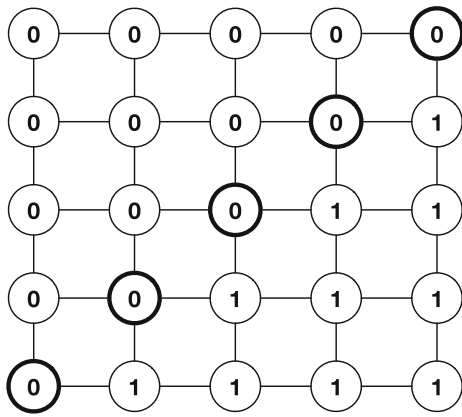
Fig. 1 An example of a correct encoding of an LBA execution in a grid; the horizontal dimension represents time, while the vertical dimension represents the position on the tape. The LBA executes a unary counter, and the marked nodes represent the position of the head

instances to prove a lower bound on the time complexity of our LCLs, and we will prove that adding all the other instances does not make the problem harder.

A simplified example. For example, consider an LBA M that encodes a unary counter, starting with the all-0 bit string and terminating when the all-1 bit string is reached. Clearly, the running time of M is linear in the length of the tape. We can represent the full execution of M using 2 dimensions, one for the tape and the other for time, and we can encode this execution on a 2-dimensional grid. See Fig. 1 for an illustration. Notice that the length of the *time* dimension of this grid depends only on the length of the *tape* dimension and on M, and for a unary counter the length of the time dimension will always be the same as the length of the tape dimension. The LCL Π_M^2 will be defined such that valid instances are 2-dimensional grids with balanced dimensions $\sqrt{n} \times \sqrt{n}$ (n nodes in total), and the idea is that, given such a grid, nodes are required to output an encoding of the full execution of M, and this would require $\Theta(n^{1/2})$ rounds (since, in order to determine their output bit, certain nodes will have to determine the bit string they are part of, which in turn requires seeing the far end of the grid where the all-0 bit string is encoded).

In order to obtain LCLs with time complexity $\Theta(n^\alpha)$, where $1/2 < \alpha < 1$, we define valid instances in a slightly different manner. We consider grids with $i > 2$ dimensions, and we let nodes choose where to encode the execution of M. We will allow nodes to choose an arbitrary dimension to use as the tape dimension, but, for technical reasons, we restrict nodes to use dimension 1 as the time dimension. The idea here is that, if the size of dimensions $2, \dots, i$ is not the same, nodes can minimize their running time by coordinating and picking the dimension j (different from 1) of smallest length as the tape dimension, and encode the execution of M using dimension 1 for time and dimension j for the tape.

Thereby we ensure that in a worst-case instance all dimensions except dimension 1 have the same length. Also, if a grid has strictly fewer or more than i dimensions, it will not be part of the family of valid instances. In other words, our LCLs can be solved faster if the input graph does not look like a multidimensional grid with the correct number i of dimensions. Then, by using LBAs with different running times, and by choosing different values for i, we will prove our claim.
Technicalities. The process of defining an LCL that requires a solution to encode an LBA execution as output needs a lot of care. Denote with M an LBA. At a high level, our LCL problems are designed such that the encoding of the execution of M needs to be done only on valid instances. In other words, the LCL Π_M^i will satisfy the following:

- If the graph looks like a multidimensional grid with i dimensions, then the output of some nodes that are part of some 2-dimensional surface of the grid must properly encode the execution of M.
- Otherwise, nodes are exempted from encoding the execution of M, but they must produce as output a proof of the fact that the instance is invalid.

This second point is somehow delicate, as nodes may try to "cheat" by claiming that a valid instance is invalid. Also, recall that in an LCL it has to be possible to check the validity of a solution *locally*, that is, there must exist a constant time distributed algorithm such that, given a correct solution, it outputs *accept* on all nodes, while given an invalid solution, it outputs *reject* on at least one node. To deal with these issues, we define our LCLs as follows:

- Valid instances are multidimensional grids with inputs. This input is a locally checkable proof that the given instance is a valid one, that is, nodes can check in constant time if the input is valid, and if it is not valid, at least one node must detect an error (for more on locally checkable proofs we refer the reader to [14]). On these valid instances, nodes can only output a correct encoding of M.
- On invalid instances, nodes must be able to prove that the input does not correctly encode a proof of the validity of the instance. This new proof must also be locally checkable.

Thus, we will define two kinds of locally checkable proofs (each using only a constant number of labels, since we will later need to encode them in the definition of the LCLs): the first is given as input to nodes, and it should satisfy that only on valid instances all nodes see a correct proof, while on invalid instances at least one node sees some error; the second is given as output from nodes, and it should satisfy

that all nodes see a correct proof only if there exists a node that, in constant time, notices some error in the first proof.

Hence, we will define LCLs that are solvable on any graph by either encoding the execution of the LBA, or by proving that the current graph is not a valid instance, where this last part is done by showing that the proof given as input is invalid on at least one node.

3.1 Roadmap

We will now describe the high level structure of this section. We will start by formally introducing Linear Bounded Automata in Sect. 3.2.

We will then introduce multidimensional grids in Sect. 3.3: these will be the hardest instances for our LCLs. These grids will be labelled with some input, and we will provide a set of local constraints for this input such that, if these constraints are satisfied for all nodes, then the graph must be a multidimensional grid of some desired dimension i (or belong to a small class of grid-like graphs that we have to deal with separately). Also, for any multidimensional grid of dimension i, it should be possible to assign these inputs in a valid manner. In other words, we design a locally checkable proof mechanism for the family of graphs of our interest, and every node will be able to verify if constraints hold by just inspecting their 3-radius neighbourhood, and essentially constraints will be valid on all nodes if and only if the graph is a valid multidimensional grid (Sects. 3.3.1 and 3.3.2).

Next we will define what are valid outputs on multidimensional grids with the desired number i of dimensions. The idea is that nodes must encode the execution of some LBA M on the surface spanned by 2 dimensions. Nodes will be able to choose which dimension to use as the *tape* dimension, but they will be forced to use dimension 1 as the *time* dimension. The reason why we do not allow nodes to choose both dimensions is that, in order to obtain complexities in the $\omega(\sqrt{n})$ spectrum, we will need the time dimension to be $\omega(\sqrt{n})$, but in any grid with at least 3 dimensions, the smallest two dimensions are always $O(\sqrt{n})$. For example, consider an LBA M that encodes a unary 5-counter, that is, a list of 5 unary counters, such that when one counter overflows, the next one is incremented. The running time of M is $\Theta(B^5)$, where B is the length of the tape. The worst case instance for the problem Π_M^3 will be a 3-dimensional grid, where dimensions 2 and 3 will have equal size $n^{1/7}$ and dimension 1 will have size $n^{5/7}$. In such an instance, nodes will be required to encode the execution of M using either dimension 2 or 3 as tape dimension, and 1 as time dimension—note that the size of dimension 1 as a function of the size of dimension 2 (or 3) matches the running time of M as a function of B. Thus, the complexity of Π_M^3 will be $\Theta(n^{5/7})$, as nodes will need to see up to distance $\Theta(n^{5/7})$ in dimension 2 (or 3). If we do not force nodes to choose dimension 1 as time, then nodes can

always find two dimensions of size $O(n^{1/2})$, and we would not be able to obtain problems with complexity $\omega(n^{1/2})$.

We will start by handling grids that are *unbalanced* in a certain way, that is, where dimension 1 is too small compared to all the others (Sect. 3.3.3). In this case, deviating from the above, we allow nodes to produce some different output that can be obtained without spending much time (this is needed to ensure that our LCLs do not get too hard on very unbalanced grids). Then, we define what the outputs must be on valid grids that are not unbalanced (Sect. 3.4). Each node must produce an output such that the ensemble of the outputs of nodes encodes the execution of a certain LBA. In particular, we define a set of output labels and a set of constraints, such that the constraints are valid on all nodes if and only if the output of the nodes correctly encodes the execution of the LBA.

We define our LCLs in Sect. 3.5. We provide a set of output labels, and constraints for these labels, that nodes can use to prove that the given graph is not a valid multidimensional grid, where the idea is that nodes can produce pointers that form a directed path that must end on a node for which the grid constraints are not satisfied. Our LCL will then be defined such that nodes can either:

- produce an encoding of the execution of the given LBA, or
- prove that dimension 1 is too short, or
- prove that there is an error in the grid structure.

All this must be done carefully, ensuring that nodes cannot claim that there is an error in valid instances, and always allowing nodes to produce a proof of an error if the instance is invalid. Also, we cannot require all nodes to consistently choose one of the three options, since that may require too much time. So we must define constraints such that, for example, it is allowed for some nodes to produce a valid encoding of the execution of the LBA, and at the same time it is allowed for some other nodes to prove that there is an error in the input proof (that maybe the first group of nodes did not see).

Finally, in Sect. 3.6 we will show upper and lower bounds for our LCLs, and in Sect. 3.7 we show how these results imply the existence of a family of LCLs that have complexities in the range between $\omega(\sqrt{n})$ and $o(n)$.

Remarks To avoid confusion, we point out that we will (implicitly) present two very different distributed algorithms in this section:

- First, we define a specific LCL problem Π_M^i. Recall that any LCL problem can be interpreted as a constant-time distributed algorithm $\mathcal{A}'$ that verifies that (G, i, o) is a valid configuration. We do not give $\mathcal{A}'$ explicitly here, but

we will present a list of constraints that can be checked in constant time by each node. This is done in Sect. 3.5.1.

– Second, we prove that the distributed complexity of Π_M^i is $\Theta(t(n))$, for some t between $\omega(\sqrt{n})$ and $o(n)$. To show this, we will need a pair of matching upper and lower bounds, and to prove the upper bound, we explicitly construct a distributed algorithm $\mathcal{A}$ that solves Π_M^i in $O(t(n))$ rounds, i.e., $\mathcal{A}$ takes (G, i) as input and produces some o as output such that (G, i, o) is a valid configuration that makes $\mathcal{A}'$ happy. This is done in Sect. 3.6.1.

Note that the specific details of Π_M^i as such are not particularly interesting; the interesting part is that Π_M^i is an LCL problem (in the strict formal sense) and its distributed time complexity is between $\omega(\sqrt{n})$ and $o(n)$. As we will see in Sect. 4 such problems do not exist in trees.

3.2 Linear bounded automata

A Linear Bounded Automaton (LBA) M consists of a Turing machine that is executed on a tape with bounded size, able to recognize the boundaries of the tape [16, p. 225]. We consider a simplified version of LBAs, where the machine is initialized with an empty tape (no input is present). We describe this simplified version of LBAs as a 5-tuple $M = (Q, q_0, f, \Gamma, \delta)$, where:

– Q is a finite set of states;
– $q_0 \in Q$ is the initial state;
– $f \in Q$ is the final state;
– Γ is a finite set of tape alphabet symbols, containing a special symbol b (*blank*), and two special symbols, L and R, called *left* and *right* markers;
– $\delta \colon Q \setminus \{f\} \times \Gamma \to Q \times \Gamma \times \{-, \leftarrow, \rightarrow\}$ is the transition function.

The tape is initialized in the following way:

– the first cell contains the symbol L;
– the last cell contains the symbol R;
– all the other cells contain the symbol b.

The head is initially positioned on the cell containing the symbol L. Then, depending on the current state and the symbol present on the current position of the tape head, the machine enters a new state, writes a symbol on the current position, and moves to some direction.

In particular, we describe the transition function δ by a finite set of 5-tuples (s_0, t_0, s_1, t_1, d), where:

1. The first 2 elements specify the input:

   – s_0 indicates the current state;

   – t_0 indicates the tape content on the current head position.

2. The remaining 3 elements specify the output:

   – s_1 is the new state;
   – t_1 is the new tape content on the current head position;
   – d specifies the new position of the head:
     – '$\rightarrow$' means that the head moves to the next cell in direction towards R;
     – '$\leftarrow$' indicates that the head moves to the next cell in direction towards L;
     – '$-$' means the head does not move.

The transition function must satisfy that it cannot move the head beyond the boundaries L and R, and the special symbols L and R cannot be overwritten. If δ is not defined on the current state and tape content, the machine terminates.

By fixing a machine M and by changing the size of the tape B on which it is executed, we obtain different running times for the machine, as a function of B. We denote by $T_M(B)$ the *running time* of an LBA M on a tape of size B. For example, it is easy to design a machine M that implements a binary counter, that starts from a tape containing all 0s and terminates when the tape contains all 1s, and this machine has a running time $T_M(B) = \Theta(2^B)$.

Also, it is possible to define a *unary k-counter*, that is, a list of k unary counters (where each one counts from 0 to $B - 1$ and then overflows and starts counting from 0 again) in which when a counter overflows, the next is incremented. It is possible to obtain running times of the form $T_M(B) = \Theta(B^k)$ by carefully implementing these counters (for example by using a single tape of length B to encode all the k counters at the cost of using more machine states and tape symbols).

The reason why we consider LBAs is that they fit nicely with the LCL framework, that requires local checkability using a constant number of labels. The definition of an LBA M does not depend on the tape size, that is, the description of M is constant compared to B. Also, by encoding the execution of M using two dimensions, one for the tape and the other for time, we obtain a structure that is locally checkable: the correctness of each constant size neighbourhood of this two dimensional surface implies global correctness of the encoding.

3.3 Grid structure

In order to obtain LCLs for the general graphs setting, we need our LCLs to be defined on any (bounded-degree) graph, and not only on some family of graphs of our interest. That is, we cannot assume any promise on the family of graphs where the LCL requires to be solved. The challenge here is that we can easily encode LBAs only on grids, but not on general graphs.
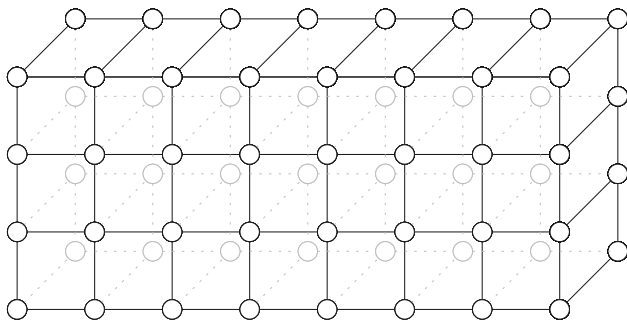
Fig. 2 An example of a 3-dimensional grid graph

Thus, we will define our LCLs in a way that there is a family of graphs, called *valid instances*, where nodes are actually required to output the encoding of the execution of a specific LBA, while on other instances nodes are exempted from doing so, but in this case they are required to prove that the graph is not a valid instance. The intuition here is that valid instances will be *hard* instances for our LCLs, meaning that when we will prove a lower bound for the time complexity of our LCLs we will use graphs that are part of the family of valid instances. Then, when we will prove upper bounds, we will show that our LCLs are always solvable, even in graphs that are invalid instances, and the time required to solve the problem in these instances is not more than the time required to solve the problem in the lower bound graphs that we provided.

We will now make a first step in defining the family of valid instances, by formally defining what a grid graph is.

Let $i \geq 2$ and $d_1, \ldots, d_i$ be positive integers. The set of nodes of an *i-dimensional grid graph* $\mathcal{G}$ consists of all i-tuples $u = (u_1, \ldots, u_i)$ with $0 \leq u_j \leq d_j$ for all $1 \leq j \leq i$. We call $u_1, \ldots, u_i$ the *coordinates* of node u and $d_1, \ldots, d_i$ the *sizes* of the *dimensions* $1, \ldots, i$. Let u and v be two arbitrary nodes of $\mathcal{G}$. There is an edge between u and v if and only if $||u - v||_1 = 1$, i.e., all coordinates of u and v are equal, except one that differs by 1. Figure 2 depicts a grid graph with 3 dimensions.

Notice that nodes do not know their position in the grid, or, for incident edges, which dimension they belong to. In fact, nodes do not even know if the graph where they currently are is actually a grid! At the beginning nodes only know the size n of the graph, and everything else must be discovered by exploring the neighbourhood.

3.3.1 Grid labels

As previously discussed, LCLs must be well defined on any (bounded-degree) graph, and we want to define our LCLs such that, if a graph is not a valid instance, then it must be easy for nodes to prove such a claim, where easy here means that the time required to produce such a proof is not larger than

the time required to encode the execution of the machine in the worst possible valid instance of the same size. For this purpose, we need to *help* nodes to produce such a proof. The idea is that a valid instance not only must be a valid grid graph, but it must also include a proof of being a valid grid graph. Thus, we will define a constant size set of labels, that will be given as input to nodes, and a set of constraints, such that, if a graph is a valid grid graph, then it is possible to assign a labelling that satisfies the constraints, while if the graph is not a valid grid graph, then it should be not possible to assign a labelling such that constraints are satisfied on all nodes (at least one node must note some inconsistency). Informally, in the LCLs that we will define, such a labelling will be provided as input to the nodes, and nodes will be able to prove that a graph is invalid by just pointing to a node where the input labelling does not satisfy these constraints.

For the sake of readability, instead of defining a set of labels and a set of local constraints that characterize valid grid graphs, we start by defining what is a valid label assignment to grid graphs, in a non-local manner. Then, we will show how to turn this to a set of locally checkable constraints. Unfortunately, we will not be able to prove that if labels satisfy these local constraints on all nodes, then the graph is actually a grid. Instead, the set of graphs that satisfy these constraints for all nodes will include a small family of additional graphs, that are graphs that locally look like a grid everywhere, but globally they are not valid grid graphs. For example, toroidal grids will be in this family. As we will show, the weaker statement that we prove will be enough for our purposes.

We now present a valid labelling for valid grid graphs. Each edge $e = \{u, v\}$ is assigned two labels $L_u(e)$ and $L_v(e)$, one for each endpoint. Label $L_u(e)$ is chosen as follows:

- $L_u(e) = (\mathsf{Next}, j)$ if $v_j - u_j = 1$;
- $L_u(e) = (\mathsf{Prev}, j)$ if $u_j - v_j = 1$.

Label $L_v(e)$ is chosen analogously. We define $\mathcal{L}^{\mathsf{grid}}$ to be the set of all possible input labels, i.e.,

$$\mathcal{L}^{\mathsf{grid}} = \big\{ (\mathsf{Prev}, j) \mid 1 \leq j \leq i \big\} \cup \big\{ (\mathsf{Next}, j) \mid 1 \leq j \leq i \big\}.$$

If we want to focus on a specific label of some edge e and it is clear from the context which of the two edge labels is considered, we may refer to it simply as the label of e.

We call the unique node that does not have any incident edge labelled (Prev, j) *origin*. Equivalently, we could define the origin directly as the node $(0, 0, \ldots, 0)$, but we want to emphasize that each node of a grid graph can infer whether it is the origin, simply by checking its incident labels.

In Sect. 3.5.1, the defined grid labels will appear as edge input labels in the definition of the new LCL problems we design. In the formal definition of an LCL problem (see

Sect. 2.2), input labels are assigned to *nodes*; however this is not a problem: that we label *edges* in our grid graphs is just a matter of convenience; we could equally well assign the labels to nodes instead of edges (and, for that matter, combine all labels of a node into a single label). The same holds for the *output* labels that are part of the definitions of the LCL problems in Sect. 3.5.1. Furthermore, we could also equally well encode the labels in the graph structure. Hence, all new time complexities presented in Sect. 3.7 can also be achieved by LCL problems without input labels (a family of problems frequently considered in the LOCAL model literature). From now on, with *grid graph* we denote a grid graph *with a valid labelling*.

3.3.2 Local checkability

As previously discussed, we want to make sure that if the graph is not a valid grid graph, then at least one node can detect this issue in constant time. Hence, we are interested in a *local characterisation* of grid graphs. Given such a characterisation, each node can check locally whether the input graph has a valid grid structure in its neighbourhood. As it turns out, such a characterization is not possible, since there are non-grid graphs that look like grid graphs locally everywhere, but we can come sufficiently close for our purposes. In fact, we will specify a set of local constraints that characterise a class of graphs that contains all grid graphs of dimension i (and a few other graphs). All the constraints depend on the 3-radius neighbourhood of the nodes, so for each input graph not contained in the characterised graph class, at least one node can detect in 3 rounds (in the LOCAL model) that the graph is not a grid graph.

For any node v and any sequence $L_1, L_2, \ldots, L_p$ of edge labels, let $z_v(L_1, L_2, \ldots, L_p)$ denote the node reached by starting in v and following the edges with labels $L_1, L_2, \ldots, L_p$. If at any point during traversing these edges there are 0 or at least 2 edges with the currently processed label, $z_v(L_1, L_2, \ldots, L_p)$ is not defined (this may happen, since nodes need to be able to check if the constraints hold also on graphs that are invalid grid graphs). Let $i \geq 2$. The full constraints are given below:

1. Basic properties of the labelling. For each node v the following hold:

   – Each edge e incident to v has exactly one (v-sided) label $L_v(e)$, and for some $1 \leq j \leq i$ we have

   $$L_v(e) \in \big\{(\mathsf{Prev}, j), (\mathsf{Next}, j)\big\}.$$

   – For any two edges e, e' incident to v, we have

   $$L_v(e) \neq L_v(e').$$

   – For any $1 \leq j \leq i$, there is at least one edge e incident to v with

   $$L_v(e) \in \big\{(\mathsf{Prev}, j), (\mathsf{Next}, j)\big\}.$$

2. Validity of the grid structure. For each node v the following hold:

   – For any incident edge $e = \{v, u\}$, we have that

   $$L_u(e) = (\mathsf{Prev}, j) \text{ if } L_v(e) = (\mathsf{Next}, j),$$
   $$L_u(e) = (\mathsf{Next}, j) \text{ if } L_v(e) = (\mathsf{Prev}, j).$$

   – Let $1 \leq j, k \leq i$ such that $j \neq k$. Also, let $e = \{v, u\}$ and $e' = \{v, u'\}$ be edges with the following v-sided labels:

   $$L_v(e) \in \big\{(\mathsf{Prev}, j), (\mathsf{Next}, j)\big\},$$
   $$L_v(e') \in \big\{(\mathsf{Prev}, k), (\mathsf{Next}, k)\big\}.$$

   Then node u has an incident edge e'' with label $L_u(e'') = L_v(e')$, and u' has an incident edge e''' with label $L_{u'}(e''') = L_v(e)$. Moreover, the two other endpoints of e'' and e''' are the same node, i.e., $z_u(L_u(e'')) = z_{u'}(L_{u'}(e'''))$.

It is clear that i-dimensional grid graphs satisfy the given constraints, but as observed above, the converse statement is not true. (As a side note for the curious reader, we mention that the converse statement can be transformed into a correct (and slightly weaker) statement by adding the small (non-local) condition that the considered graph contains a node not having any incident edge labelled with some (Prev, j), for all dimensions j. However, due to its non-local nature, we cannot make use of such a condition.)

3.3.3 Unbalanced grid graphs

In Sect. 3.3.2, we saw the basic idea behind ensuring that non-grid graphs are not among the hardest instances for the LCL problems we construct. In this section, we will study the ingredient of our LCL construction that guarantees that grid graphs where the dimensions have "wrong" sizes are not worst-case instances. More precisely, we want that the hardest instances for our LCL problems are grid graphs with the property that there is at least one dimension $2 \leq j \leq i$ whose size is not larger than the size of dimension 1. In the following, we will show how to make sure that *unbalanced* grid graphs, i.e., grid graphs that do not have this property, allow nodes to find a valid output without having to see too far. In a sense, in the LCLs that we construct, one possible valid output is to produce a proof that the grid is unbalanced in a wrong way, and since the validity of an output assignment

for an LCL must be locally checkable, we want such a proof to be locally checkable as well.

Thus, in the LCLs that we will define, nodes of an arbitrary graph will be provided with some input labelling that encodes a (possibly wrong) proof that claims that the current graph is a valid grid graph. Then, if the graph does not look like a grid, nodes can produce a locally checkable proof that claims that this input proof is wrong. Instead, if the graph does look like a grid, but this grid appears to be unbalanced in some undesired way, nodes can produce a locally checkable proof about this fact.

More formally, consider a grid graph with i dimensions of sizes $d_1, \ldots, d_i$. If $d_1 < d_j$ for all $2 \le j \le i$, the following output labelling is regarded as correct in any constructed LCL problem:

- For all $0 \le t \le d_1$, node $v = (v_1, \ldots, v_i)$ satisfying $v_1 = \cdots = v_i = t$ is labelled Unbalanced.
- All other nodes are labelled Exempt$_U$.

This labelling is clearly locally checkable, i.e., it can be described as a collection of local constraints: Each node v labelled Unbalanced checks that

1. its two "diagonal neighbours"

    $u = z_v((\text{Prev}, 1), (\text{Prev}, 2), \ldots, (\text{Prev}, i))$ and
    $w = z_v((\text{Next}, 1), (\text{Next}, 2), \ldots, (\text{Next}, i))$,

    both exist (i.e., are defined) and are both labelled Unbalanced, or
2. w exists and is labelled Unbalanced and v has no incident edge labelled (Prev, j), or
3. u exists and is labelled Unbalanced and v has an incident edge labelled (Next, j) for all $2 \le j \le i$, but no incident edge labelled $(\text{Next}, 1)$.

Condition 3 ensures that the described diagonal chain of Unbalanced labels terminates at a node whose first coordinate is d_1 (i.e., the maximal possible value for the coordinate corresponding to dimension 1), but whose second, third, …, coordinate is strictly smaller than $d_2, d_3, \ldots$, respectively, thereby guaranteeing that grid graphs that are not unbalanced do not allow the output labelling specified above. Finally, the origin checks that it is labelled Unbalanced, in order to prevent the possibility that each node simply outputs Exempt$_U$. We refer to Fig. 3 for an example of an unbalanced 2-dimensional grid and its labelling. We define $\mathcal{L}^{\text{unbalanced}}$ to be the set containing {Unbalanced, Exempt$_U$}.
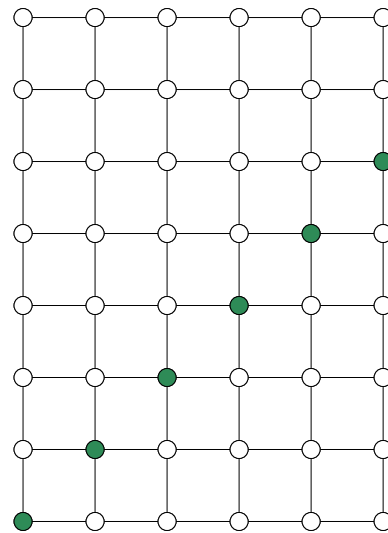

Fig. 3 An example of an unbalanced grid with 2 dimensions; the nodes in green are labelled with Unbalanced, while the white nodes are labelled with Exempt$_U$

3.4 Machine encoding

After examining the cases of the input graph being a non-grid graph or an unbalanced grid graph, in this section, we turn our attention towards the last remaining case: that is the input graph is actually a grid graph for which there is a dimension with size smaller than or equal to the size of dimension 1. In this case, we require the nodes to work together to create a global output that is determined by some LBA M. Essentially, the execution of M has to be written (as node outputs) on a specific part of the grid graph. In order to formalise this relation between the desired output and the LBA M, we introduce the notion of an *M-encoding graph* in the following section.

3.4.1 Labels

Let M be an LBA, and consider the execution of M on a tape of size B. Let $S_\ell = (s_\ell, h_\ell, t_\ell)$ be the whole state of M after step ℓ, where s_ℓ is the machine internal state, h_ℓ is the position of the head, and t_ℓ is the whole tape content. The content of the cell in position $y \in \{0, \ldots, B-1\}$ after step ℓ is denoted by $t_\ell[y]$. We denote by $(x, y)_k$ the node $v = (v_1, \ldots, v_i)$ having $v_1 = x$, $v_k = y$, and $v_j = 0$ for all $j \notin \{1, k\}$. An (output-labelled) grid graph of dimension i is an *M-encoding graph* if there exist a tape size B and a dimension $2 \le k \le i$ satisfying the following conditions.

(C1) d_k is equal to $B - 1$.
(C2) For all $0 \le x \le \min\{T_M(B), d_1\}$ and all $0 \le y \le B - 1$, it holds that:

   (a) Node $(x, y)_k$ is labelled with Tape$(t_x[y])$.

(b) Node $(x, y)_k$ is labelled with $\mathsf{State}(s_x)$.

(c) Node $(x, h_x)_k$ is labelled with Head.

(d) Node $(x, y)_k$ is labelled with $\mathsf{Dimension}(k)$.

(C3) All other nodes are labelled with Exempt_M.

Intuitively, the 2-dimensional surface expanding in dimensions 1 and k (having all the other coordinates equal to 0), encodes the execution of M. More precisely, the nodes $(x, 0)_k, (x, 1)_k, \ldots, (x, B - 1)_k$ together represent the state of the tape at time x, i.e., dimension 1 constitutes the time axis whereas the tape itself is unrolled along dimension k. In particular, the nodes $(0, 1)_k, (0, 2)_k, \ldots, (0, B - 2)_k$ representing the (inner part of the) tape at the beginning of the computation are all labelled with the blank symbol b (or, if we want to be very precise, $\mathsf{Tape}(b)$), the nodes $(0, 0)_k, (0, 1)_k, \ldots$ representing the left end of the tape (at different points in time) are labelled with the left marker L, and the nodes $(B - 1, 0)_k, (B - 1, 1)_k, \ldots$ representing the right end of the tape are labelled with the right marker R. We define $\mathcal{L}^{\text{encoding}}$ to be the set of all possible output labels used to define an M-encoding graph.

3.4.2 Local checkability

In order to force nodes to output labels that turn the input grid graph into an M-encoding graph, we must be able to describe Conditions (C1)–(C3) in the form required by an LCL problem, i.e., as locally checkable constraints. In the following, we provide such a description, showing that the nodes can check whether the graph correctly encodes the execution of a given LBA M.

Constraint (LC1): Each node v is labelled with either Exempt_M or $\mathsf{Dimension}(k)$ for exactly one $2 \leq k \leq i$. In the former case, node v has no other labels, in the latter case, v additionally has some Tape and some State label, and potentially the label Head, but no other labels.

Constraint (LC2): The origin has label $\mathsf{Dimension}(k)$, for some $2 \leq k \leq i$.

Constraint (LC3): If a node v labelled $\mathsf{Dimension}(k)$ for some $2 \leq k \leq i$ has an incident edge e labelled with $L_v(e) = (\mathsf{Prev}, j)$, then $j = 1$ or $j = k$. Moreover, for each node v labelled $\mathsf{Dimension}(k)$, nodes $z_v((\mathsf{Prev}, 1))$, $z_v((\mathsf{Prev}, k))$ and $z_v((\mathsf{Next}, k))$ (provided they are defined) are also labelled $\mathsf{Dimension}(k)$.

Constraint (LC4): For each node v labelled $\mathsf{Dimension}(k)$ for some $2 \leq k \leq i$, the following hold:

1. If v does not have an incident edge labelled $(\mathsf{Prev}, 1)$, then

   (a) if v does not have an incident edge labelled (Prev, k), then it must have labels Head and $\mathsf{Tape}(L)$;

   (b) if v does not have an incident edge labelled (Next, k), then it must have label $\mathsf{Tape}(R)$;

   (c) if v has an incident edge labelled (Prev, k) and an incident edge labelled (Next, k), then it has label $\mathsf{Tape}(b)$;

   (d) v has label $\mathsf{State}(q_0)$;

   (e) if $q_0 \neq f$, then node $z_v((\mathsf{Next}, 1))$ (if defined) is labelled $\mathsf{Dimension}(k)$.

2. If v has an incident edge labelled $(\mathsf{Prev}, 1)$, v has labels $\mathsf{State}(q)$ and $\mathsf{Tape}(t)$, and $z_v((\mathsf{Prev}, 1))$ has labels $\mathsf{State}(q')$ and $\mathsf{Tape}(t')$, then

   (f) $q' \neq f$;

   (g) if $z_v((\mathsf{Prev}, 1))$ is labelled with Head, then q and t are derived from q' and t' according to the specifications of the LBA M, and the new position of the head is either on v itself, or on $z_v((\mathsf{Prev}, k))$, or on $z_v((\mathsf{Next}, k))$, depending on M;

   (h) otherwise, $t = t'$ and the nodes $z_v((\mathsf{Prev}, k))$ and $z_v((\mathsf{Next}, k))$ (if defined) are labelled $\mathsf{State}(q)$;

   (i) if $q \neq f$, then node $z_v((\mathsf{Next}, 1))$ (if defined) is labelled $\mathsf{Dimension}(k)$.

Correctness. It is clear that an M-encoding graph satisfies Constraints (LC1)–(LC4). Conversely, we want to show that any graph satisfying Constraints (LC1)–(LC4) is an M-encoding graph. We start by setting $B := d_k + 1$, which already implies Condition (C1).

Constraints (LC1)–(LC3) ensure that there is a 2-dimensional surface $\mathcal{S}$ on which the execution of M is encoded: (LC3) ensures that for any node labelled $\mathsf{Dimension}(k)$, all coordinates except potentially those corresponding to dimension 1 and k are 0, i.e., each node labelled $\mathsf{Dimension}(k)$ is of the form $(x, y)_k$ for some x, y. Moreover, according to (LC3), whenever some $(x, y)_k$ is labelled $\mathsf{Dimension}(k)$, then also all $(x', y')_k$ that satisfy $x' \leq x$ are labelled $\mathsf{Dimension}(k)$ and, in particular, the origin is also labelled $\mathsf{Dimension}(k)$. Since, by (LC1), no node (in particular, the origin) is labelled $\mathsf{Dimension}(k)$ for more than one k, it follows that there is at most one k for which there exist nodes with label $\mathsf{Dimension}(k)$, and these nodes are exactly the nodes $(x, y)_k$ for which x does not exceed some threshold value (which as we will see will be exactly $\min\{T_M(B), d_1\}$). (LC2) ensures that this value is at least 1; in particular, there are nodes that are not labelled Exempt_M. (LC1) ensures that all nodes not labelled $\mathsf{Dimension}(k)$ are labelled Exempt_M.

Constraints (LC4a)–(LC4d) ensure that the LBA M is initialized correctly: (LC4a)–(LC4c) ensure that $(0, 0)_k$ is labelled with $\mathsf{Tape}(L) = \mathsf{Tape}(t_0[0])$, $(0, y)_k$ is labelled with $\mathsf{Tape}(b) = \mathsf{Tape}(t_0[y])$ for each $1 \leq y \leq B - 2$, and $(0, B - 1)_k$ is labelled with $\mathsf{Tape}(R) = \mathsf{Tape}(t_0[B - 1])$, which implies Condition (C2a) for $x = 0$. Similarly, (LC4a)

also ensures (C2c) for $x = 0$, and (LC4d) ensures (C2b) for $x = 0$.

Constraints (LC4e)–(LC4i) ensure a correct execution of each step of M, and that nodes on $\mathcal{S}$ output Exempt_M only after the termination state of M is reached: Constraints (LC4e) and (LC4i) ensure that the threshold value for y up to which all $(x, y)_k$ are labelled with $\mathsf{Dimension}(k)$ is at least $T_M(B)$, unless $T_M(B) > d_1$, in which case the threshold value is d_1. (LC4f) ensures that the threshold value does not exceed $T_M(B)$, thereby implying Conditions (C2d) and (C3). Here we use the observation derived from (LC1) that all nodes not labelled $\mathsf{Dimension}(k)$ are labelled Exempt_M. (LC4g) and (LC4h) imply that if the nodes $(x, 0)_k, (x, 1)_k, \ldots, (x, B - 1)_k$ encode the state of the computation after step x, and the corresponding machine internal state is not the final state, then also the nodes $(x + 1, 0)_k, (x + 1, 1)_k, \ldots, (x + 1, B - 1)_k$ encode the state of the computation after step $x + 1$. As we already observed above that (LC4a)–(LC4d) ensure that $(0, 0)_k, (0, 1)_k, \ldots, (0, B - 1)_k$ encode the initial state of the computation, we obtain by induction (and our obtained knowledge about the threshold value) that (C2a)–(C2c) hold for all $0 \leq x \leq \min\{T_M(B), d_1\}$.

3.5 LCL construction

Fix an integer $i \geq 2$, and let M be an LBA with running time T_M. As we do not fix a specific size of the tape, T_M can be seen as a function that maps the tape size B to the running time $T_M(B)$ of the LBA executed on a tape of size B. We now construct an LCL problem Π_M^i with complexity related to T_M. Note that Π_M^i depends on the choice of i. The general idea of the construction is that nodes can either:

- produce a valid encoding of the execution of M, or
- prove that dimension 1 is too short, or
- prove that there is an error in the (grid) graph structure.

We need to ensure that on balanced grid graphs it is not easy to claim that there is an error, while allowing an efficient solution on invalid graphs, i.e., graphs that contain a local error (some invalid label), or a global error (a grid structure that wraps, or dimension 1 too short compared to the others).

3.5.1 LCL Problem Π_M^i

Formally, we specify the LCL problem Π_M^i as follows. The input label set for Π_M^i is the set $\mathcal{L}^{\mathsf{grid}}$ of labels used in the grid labelling (see Sect. 3.3.1). The possible output labels are the following:

1. The labels from $\mathcal{L}^{\mathsf{encoding}}$ (see Sect. 3.4).
2. The labels from $\mathcal{L}^{\mathsf{unbalanced}}$ (see Sect. 3.3.3).

3. The set of *error labels* $\mathcal{L}^{\mathsf{error}}$. This set is defined to contain the *error label* Error, and *error pointers*, i.e., all possible pairs (s, r), where s is either (Next, j) or (Prev, j) for some $1 \leq j \leq i$, and $r \in \{0, 1\}$ is a bit whose purpose it is to distinguish between two different types of error pointers, *type 0* pointers and *type 1* pointers.

Intuitively, nodes that notice that there is/must be an error in the grid structure, but are not allowed to output Error because the grid structure is valid *in their local neighbourhood*, can point in the direction of an error. However, the nodes have to make sure that the error pointers form a chain that actually ends in an error. In order to make the proofs in this section more accessible, we distinguish between the two types of error pointers mentioned above; roughly speaking, type 0 pointers will be used by nodes that (during the course of the algorithm) cannot see an error in the grid structure, but notice that the grid structure wraps around in some way, while type 1 pointers are for nodes that can actually see an error. Here, with "wrapping around", we mean that there is a node v and a sequence of edge labels $L_1, L_2, \ldots, L_p$ such that

1. There exists a dimension j such that the number of labels (Prev, j) in this sequence is different from the number of labels (Next, j), and
2. $z_v(L_1, L_2, \ldots, L_p) = v$, i.e., we can walk from some node to itself without going in each dimension the same number of times in one direction as in the other.

If the grid structure wraps around, then there must be an error somewhere (and nodes that see that the grid structure wraps around know where to point their error pointer to), or following an error pointer chain results in a cycle; however, since the constraints we put on error pointer chains are *local* constraints (as we want to define an LCL problem), the global behaviour of the chain is irrelevant. We will not explicitly prove the global statements made in this informal overview; for our purposes it is sufficient to focus on the local views of nodes.

Note that if a chain of type 0 error pointers does not cycle, then at some point it will turn into a chain of type 1 error pointers, which in turn will end in an error. Chains of type 1 error pointers cannot cycle. We refer to Fig. 4 for an example of an error pointer chain.

An output labelling for problem Π_M^i is correct if the following conditions are satisfied.

1. Each node v produces at least one output label. If v produces at least two output labels, then all of v's output labels are contained in $\mathcal{L}^{\mathsf{encoding}} \setminus \{\mathsf{Exempt}_M\}$.
2. Each node at which the input labelling does not satisfy the local grid graph constraints given in Sect. 3.3.2 outputs Error. All other nodes do not output Error.
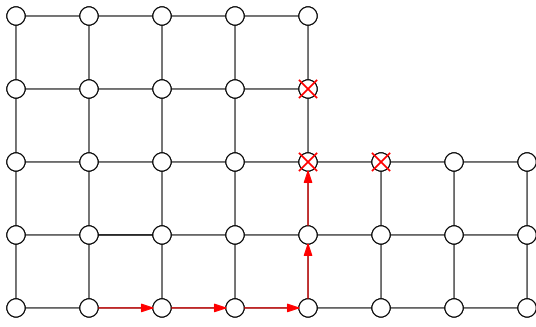
Fig. 4 An example of an error pointer chain (shown in red). Nodes that are marked with a red cross are those who actually see an error in the grid structure. The output of only some of the depicted nodes is shown (color figure online)

3. If a node v outputs $\mathsf{Exempt_U}$ or $\mathsf{Exempt_M}$, then v has at least one incident edge e with input label $L_v(e) = (\mathsf{Prev}, j)$, where $j \in \{1, \ldots, i\}$.
4. If the output labels of a node v are contained in $\mathcal{L}^{\mathsf{encoding}} \setminus \{\mathsf{Exempt_M}\}$, then either there is a node in v's 2-radius neighbourhood that outputs a label from $\mathcal{L}^{\mathsf{error}}$, or the output labels of all nodes in v's 2-radius neighbourhood are contained in $\mathcal{L}^{\mathsf{encoding}}$. Moreover, in the latter case v's 2-radius neighbourhood has a valid grid structure and the local constraints of an M-encoding graph, given in Sect. 3.4.2, are satisfied at v.
5. If the output of a node v is $\mathsf{Unbalanced}$, then either there is a node in v's i-radius neighbourhood that outputs a label from $\mathcal{L}^{\mathsf{error}}$, or the output labels of all nodes in v's i-radius neighbourhood are contained in $\mathcal{L}^{\mathsf{unbalanced}}$. Moreover, in the latter case v's i-radius neighbourhood has a valid grid structure and the local constraints for a proof of unbalance, given in Sect. 3.3.3, are satisfied at v.
6. Let v be a node that outputs an error pointer (s, r). Then $z_v(s)$ is defined, i.e., there is exactly one edge incident to v with input label s. Let u be the neighbour reached by following this edge from v, i.e., $u = z_v(s)$. Then u outputs either Error or an error pointer (s', r'), where in the latter case the following hold:

   - $r' \geq r$, i.e., the type of the pointer cannot decrease when following a chain of error pointers;
   - if $r' = 0 = r$, then $s' = s$, i.e., the pointers in a chain of error pointers of type 0 are consistently oriented;
   - if $r' = 1 = r$ and

     $$s \in \{(\mathsf{Prev}, j), (\mathsf{Next}, j)\},$$
     $$s' \in \{(\mathsf{Prev}, j'), (\mathsf{Next}, j')\},$$

     then $j' \geq j$, i.e., when following a chain of error pointers of type 1, the dimension of the pointer cannot decrease;

   - if $r' = 1 = r$ and

     $$s, s' \in \{(\mathsf{Prev}, j), (\mathsf{Next}, j)\}$$

     for some $1 \leq j \leq i$, then $s' = s$, i.e., any two subsequent pointers in the same dimension have the same direction.

These conditions are clearly locally checkable, so Π_M^i is a valid LCL problem.

3.6 Time complexity

Let M be an LBA, $i \geq 2$ an integer, and B the smallest positive integer satisfying $n \leq B^{i-1} \cdot T_M(B)$. We will only consider LBAs M with the property that $B \leq T_M(B)$ and for any two tape sizes $B_1 \geq B_2$ we have $T_M(B_1) \geq T_M(B_2)$. In the following, we prove that Π_M^i has time complexity $\Theta(n/B^{i-1}) = \Theta(T_M(B))$.

3.6.1 Upper bound

In order to show that Π_M^i can be solved in $O(T_M(B))$ rounds, we provide an algorithm $\mathcal{A}$ for Π_M^i. Subsequently, we prove its correctness and that its running time is indeed $O(T_M(B))$. Algorithm $\mathcal{A}$ proceeds as follows.

First, each node v gathers its constant-radius neighbourhood, and checks whether there is a local error in the grid structure at v, i.e., if constraints given in Sect. 3.3.2 are not satisfied. In that case, v outputs Error. Then, each node v that did not output Error gathers its R-radius neighbourhood, where $R = c \cdot T_M(B)$ for a large enough constant $c \geq i$, and acts according to the following rules.

- If there is a node labelled Error in v's R-radius neighbourhood, then v outputs an error pointer $(s, 1)$ of type 1, where $s \in \{(\mathsf{Prev}, j), (\mathsf{Next}, j)\}$ has the following property: among all shortest paths from v to some node that outputs Error, there is one where the first edge e on the path has input label $L_v(e) = s$, but, for any $j' < j$, there is none where the first edge e has input label $L_v(e) \in \{(\mathsf{Prev}, j'), (\mathsf{Next}, j')\}$.
- Now consider the case that there is no node labelled Error in v's R-radius neighbourhood, but there is a path P from v to itself with the following property: Let $\mathcal{L}'$ be the sequence of labels read on the edges when traversing P, where for each edge $e = \{u, w\}$ traversed from u to w we only read the label $L_u(e)$. Then there is some $1 \leq j \leq i$ such that the number of occurrences of label (Prev, j) in $\mathcal{L}'$ is not the same as the number of occurrences of label (Next, j) in $\mathcal{L}'$. (In other words, the grid structure wraps around in some way.) Let k be the small-

est j for which such a path P exists. Then v outputs an error pointer $(s, 0)$ of type 0, where $s = (\mathsf{Next}, k)$.

- If the previous two cases do not apply (i.e., the input graph has a valid grid structure and does not wrap around, as far as v can see), then v checks for each dimension $1 \leq j \leq i$ whether in v's R-radius neighbourhood there is both a node that does not have an incident edge labelled (Prev, j) and a node that does not have an incident edge labelled (Next, j). (As we allow arbitrary input graphs, there could be several such node pairs.) For each dimension j for which such two nodes exist, v computes the size d_j of the dimension by determining the distance between those two nodes w.r.t. dimension j, i.e., the absolute difference of the j-th coordinates of the two nodes. (Note that v does not know the absolute coordinates, but can assign coordinates to the nodes it sees in a locally consistent manner, and that the absolute difference of the coordinates of those nodes does not depend on v's choice as long as it is consistent.) Here, and in the following, v assumes that the input graph also continues to be a grid graph outside of v's R-radius neighbourhood. Then, v checks whether among these j there is a dimension $2 \leq j' \leq i$ with $d_{j'} \leq T_M(B)$ that, in case v actually computed the size of dimension 1, also satisfies $d_{j'} \leq d_1$. Now there are two cases:

  1. If such a j' exists, then v chooses the smallest such j' (breaking ties in a consistent manner), denoted by k, and computes its coordinate in dimension k. Node v also computes its coordinate in dimension 1 or verifies that it is larger than $T_M(B)$. Since v can determine whether it has coordinate 0 in all the other dimensions, it has all the information required to compute its output labels in the M-encoding graph where the execution of M takes place on the surface that expands in dimensions 1 and k. Consequently, v outputs these labels (that is, labels from $\mathcal{L}^{\mathsf{encoding}}$, defined in Sect. 3.4). Note further that, according to the definition of an M-encoding graph, v outputs Exempt_M if it verifies that its coordinate in dimension 1 is larger than $T_M(B)$, even if it has coordinate 0 in all dimensions except dimension 1 and (possibly) k. Note that if the input graph does not continue to be a grid graph outside of v's R-radius neighbourhood, then neighbours of v might output error pointers, but this is still consistent with the local constraints of Π_M^i.
  2. If no such j' exists, then, by the definition of B, node v sees (nodes at) both borders of dimension 1. In this case, v can compute the label it would output in a proof of unbalance (that is, a label from $\mathcal{L}^{\mathsf{unbalanced}}$, defined in Sect. 3.3.3), since for this, v only has to determine whether its coordinates are the same in all

dimensions (which is possible as all nodes with this property are in distance at most $i \cdot T_M(B)$ from the origin). Consequently, v outputs this label. Again, if the input graph does not continue to be a grid graph outside of v's R-radius neighbourhood, then, similarly to the previous case, the local constraints of Π_M^i are still satisfied.

Theorem 1 *Problem Π_M^i can be solved in $O(T_M(B))$ rounds.*

Proof We will show that algorithm $\mathcal{A}$ solves problem Π_M^i in $O(T_M(B))$ rounds. It is easy to see that the complexity of $\mathcal{A}$ is $O(T_M(B))$. We need to prove that it produces a valid output labelling for Π_M^i. For this, first consider the case that the input graph is a grid graph. Let $2 \leq k \leq i$ be the dimension with minimum size (apart, possibly, from the size of dimension 1). If $d_k \leq d_1$, then $d_k \leq T_M(B)$, by the definition of B and the assumption that $T_M(B) \geq B$. In this case, according to algorithm $\mathcal{A}$, the nodes output labels that turn the input graph into an M-encoding graph, thereby satisfying the local constraints of Π_M^i. If, on the other hand, $d_k > d_1$, then according to algorithm $\mathcal{A}$, the nodes output labels that constitute a valid proof for unbalanced grids, again ensuring that the local constraints of Π_M^i are satisfied.

If the input graph looks like a grid graph from the perspective of some node v (but might not be a grid graph from a global perspective), then there are two possibilities: either the input graph also looks like a grid graph from the perspective of all nodes in v's 2-radius neighbourhood, in which case the above arguments ensure that the local constraints of Π_M^i (regarding M-encoding labels, i.e., labels from $\mathcal{L}^{\mathsf{encoding}}$) are satisfied at v, or some node in v's 2-radius neighbourhood notices that the input graph is not a grid graph, in which case it outputs an error pointer and thereby ensures the local correctness of v's output. The same argument holds for the local constraints of Π_M^i regarding labels for proving unbalance (instead of labels from $\mathcal{L}^{\mathsf{encoding}}$), with the only difference that in this case we have to consider v's i-radius neighbourhood (instead of v's 2-radius neighbourhood).

What remains to show is that the constraints of Π_M^i are satisfied at nodes v that output Error or an error pointer. If v outputs Error according to $\mathcal{A}$, then the constraints of Π_M^i are clearly satisfied, hence assume that v outputs an error pointer (s, r).

We first consider the case that $r = 0$, i.e., v outputs an error pointer of type 0. In this case, according to the specifications of $\mathcal{A}$, there is no error in the grid structure in v's R-radius neighbourhood. Let u be the neighbour of v the error pointer points to, i.e., the node reached by following the edge with label s from v. Due to the valid grid structure around v, node u is well-defined. According to the specification of Π_M^i, we have to show that u outputs an error pointer (r', s') satisfying $r' = 1$ or $s' = s$. If there is a node in u's R-radius neigh-

bourhood that outputs Error, then u outputs an error pointer of type 1, i.e., $r' = 1$. Thus, assume that there is no such node, which implies that the grid structure in u's R-radius neighbourhood is valid as well.

Consider a path from v to itself inside v's R-radius neighbourhood, and let $L_1, \ldots, L_h$ be the sequence of edge labels read when traversing this path, where for each edge $e = \{w, x\}$, we only consider the input label that belongs to the node from which the traversal of the edge starts, i.e., $L_w(e)$ if edge e is traversed from w to x. Then, due to the grid structure of v's R-radius neighbourhood, there is such a path P with the following property: for each $1 \leq j \leq i$, at most one of (Prev, j) and (Next, j) is contained in the edge label sequence (as any two labels (Prev, j) and (Next, j) "cancel out"), and the edge label sequence (and thus the directions of the edges) is ordered non-decreasingly w.r.t. dimension, i.e., if $L_{h'} \in \{(\text{Prev}, j'), (\text{Next}, j')\}$ and $L_{h''} \in \{(\text{Prev}, j''), (\text{Next}, j'')\}$ for some $1 \leq h' \leq h'' \leq h$, then $j' \leq j''$. Also, we can assume that the edge label L_1 of the first edge on P is of the kind (Next, j) for some j as we can reverse the direction of path P and subsequently transform it into a path with the above properties by reordering the edge labels. Due to the specification of $\mathcal{A}$ regarding type 0 error pointer outputs and the above observations, we can assume that $L_1 = s$.

Consider the path P' obtained by starting at u and following the edge label sequence $L_2, \ldots, L_h, s$. Since $u = z_v(L_1) = z_v(s)$ and $v = z_v(L_1, \ldots, L_h)$, we have that $u = z_u(L_2, \ldots, L_h, s)$. Since P is contained in the R-radius neighbourhood of v (and P has the nice structure outlined above), P' is contained in the R-radius neighbourhood of u, thereby ensuring that u outputs a type 0 error pointer. Let k and k' be the indices satisfying (Next, k) $= s$ and (Next, k') $= s'$, respectively. Again due to the specification of $\mathcal{A}$ regarding type 0 error pointer outputs, we see that $k' \leq k$. However, using symmetric arguments to the ones provided above, it is also true that for each path P'' from u to itself of the kind specified above, there is a path from v to itself that contains the same labels in the label sequence as P'' (although not necessarily in the same order), which implies that $k \leq k'$. Hence, $k' = k$, and we obtain $s' = s$, as required.

Now consider the last remaining case, i.e., that v outputs an error pointer $(s, 1)$ of type 1. Again, let u be the neighbour of v the error pointer points to, i.e., the node reached by following the edge with label s from v. Let D and D' be the lengths of the shortest paths from v, resp. u to some node that outputs Error. By the specification of $\mathcal{A}$ regarding type 1 error pointer outputs, we know that $D' = D - 1$, which ensures that u outputs Error or an error pointer of type 1. If u outputs Error, then the local constraints of Π_M^i are clearly satisfied at v. Thus, consider the case that u outputs an error pointer $(s', 1)$ of type 1. Let k and k' be the indices satisfying

$s \in \{(\text{Prev}, k), (\text{Next}, k)\}$ and $s' \in \{(\text{Prev}, k'), (\text{Next}, k')\}$, respectively. We need to show that either $k' = k$ and $s' = s$, or $k' > k$.

Suppose for a contradiction that either $k' = k$ and $s' \neq s$, or $k' < k$. Note that the latter case also implies $s' \neq s$. Consider a path P' of length D' from u to some node w outputting Error with the property that the first edge e' on P' has input label s'. Such a path P' exists by the specification of $\mathcal{A}'$. Let P be the path from v to w obtained by appending P' to the path from v to u consisting of edge $e = \{v, u\}$. Note that $L_v(e) = s$. Since v did not output Error, the local grid graph constraints, given in Sect. 3.3.2, are satisfied at v. Hence, if $k' < k$, we can obtain a path P'' from v to w by exchanging the directions of the first two edges of P, i.e., P'' is obtained from P by replacing the first two edges e, e' by the edges $e'' = \{v, z_v(s')\}$, $e''' = \{z_v(s'), z_u(s')\}$. Note that $L_v(e'') = s'$ and $L_{z_v(s')}(e''') = s$. In this case, since P'' has length D and starts with an edge labelled s', we obtain a contradiction to the specification of $\mathcal{A}$ regarding error pointers of type 1, by the definitions of k, k', s, s', D. Thus assume that $k' = k$ and $s' \neq s$. In this case, $z_v(s, s') = z_u(s') = v$ which implies that $D' = D + 1$, by the definitions of D, D', P'. This is a contradiction to the equation $D' = D - 1$ observed above. Hence, the local constraints of Π_M^i are satisfied at v. □

3.6.2 Lower bound

Theorem 2 *Problem Π_M^i cannot be solved in $o(T_M(B))$ rounds.*

Proof Consider i-dimensional grid graphs where the number n of nodes satisfies $n = B^{i-1} \cdot T_M(B)$. Clearly, there are infinitely many n with this property, due to the definition of B. More specifically, consider such a grid graph $\mathcal{G}$ satisfying $d_j = B$ for all $j \in \{2, \ldots, i\}$, and $d_1 = T_M(B)$. By the local constraints of Π_M^i, the only valid global output is to produce an M-encoding graph, on a surface expanding in dimensions 1 and k for some $k \in \{2, \ldots, i\}$. In fact:

– If nodes try to prove that the grid graph is unbalanced, since $T_M(B) \geq B$, the proof must either be locally wrong, or, if nodes outputting Unbalanced actually form a diagonal chain, this chain must terminate on a node that, for any $2 \leq j \leq i$, does not have an incident edge labelled (Next, j), that is, constraints defined in Sect. 3.3.3 are not satisfied, which also violates the local constraints of Π_M^i.

– If nodes try to produce an error pointer, since the specification of the validity of pointer outputs in the local constraints of Π_M^i ensures that on grid graphs a pointer chain cannot visit any node twice, any error pointer chain must terminate somewhere. Since no nodes can be labelled Error, this is not valid.

– The only remaining possibility for the origin is to output a label from $\mathcal{L}^{\text{encoding}}\setminus\{\text{Exempt}_\mathsf{M}\}$, which already implies that all the other nodes must produce outputs that turn the graph into an M-encoding graph.

Thus, it remains to show that producing a valid M-encoding labelling requires time $\Omega(T_M(B))$. Consider the node having coordinate 1 equal to $x = T_M(B)$ and all other coordinates equal to 0. This node must be labelled $\text{State}(f)$, the nodes with coordinate 1 strictly less than x must not be labelled $\text{State}(f)$, and the nodes with coordinate 1 strictly greater than x must be labelled Exempt_M. Thus, a node needs to know if it is at distance $T_M(B)$ from the boundary of coordinate 1, which requires $\Omega(T_M(B))$ time. $\qquad\square$

3.7 Instantiating the LCL construction

Our construction is quite general and allows to use a wide variety of LBAs to obtain many different LCL complexities. As a proof of concept, in Theorems 3 and 4, we show some complexities that can be obtained using some specific LBAs. Recall that

– if we choose our LBA M to be a unary k-counter, for constant k, then M has a running time of $T_M(B) = \Theta(B^k)$, and
– if we choose M to be a binary counter, then M has a running time of $T_M(B) = \Theta(2^B)$.

Theorem 3 *For any rational number $0 \le \alpha \le 1$, there exists an* LCL *problem with time complexity $\Theta(n^\alpha)$.*

Proof Let $j > k$ be positive integers satisfying $\alpha = k/j$. Given an LBA M with running time $\Theta(B^k)$ and choosing $i = j-k+1$, we obtain an LCL problem Π_M^i with complexity $\Theta(n/B^{j-k})$. We have that $n = \Theta(B^{j-k} \cdot T_M(B)) = \Theta(B^j)$, which implies $B = \Theta(n^{1/j})$. Thus the time complexity of Π_M^i is $\Theta(n/n^{(j-k)/j}) = \Theta(n^\alpha)$. $\qquad\square$

Theorem 4 *There exist* LCL *problems of complexities $\Theta(n/\log^j n)$, for any positive integer j.*

Proof Given an LBA M with running time $\Theta(2^B)$ and choosing $i = j+1$, we obtain an LCL problem Π_M^i with complexity $\Theta(n/B^j)$. We have that $n = \Theta(B^j \cdot T_M(B)) = \Theta(B^j \cdot 2^B)$, which implies $B = \Theta(\log n)$. Thus the time complexity of Π_M^i is $\Theta(n/\log^j n)$. $\qquad\square$

4 Complexity gap on trees

In the previous section, we have seen that there are infinite families of LCLs with distinct time complexities between $\omega(\sqrt{n})$ and $o(n)$. In this section we prove that on trees there

are no such LCLs. That is, we show that if an LCL is solvable in $o(n)$ rounds on trees, it can be also solved in $O(\sqrt{n})$ rounds.

The high level idea is the following. Consider an LCL Π that can be solved in $o(n)$ rounds on a tree T of n nodes, that is, there exists a distributed algorithm $\mathcal{A}$, that, running on each node of T, outputs a valid labelling for Π in sublinear time. We show how to speed this algorithm up, and obtain a new algorithm $\mathcal{A}'$ that runs in $O(\sqrt{n})$ rounds and solves Π as well.

To do this, we show that nodes of T can distributedly construct, in $O(\sqrt{n})$ rounds, a virtual graph S of size $N \gg n$. This graph S will be defined such that we can run $\mathcal{A}$ on it, and use the solution that we get to obtain a solution for Π on T. Moreover, for each node of T, it will be possible to simulate the execution of $\mathcal{A}$ on S by just inspecting its neighbourhood of radius $O(\sqrt{n})$ on T, thus obtaining an algorithm for Π running in $O(\sqrt{n})$ rounds.

We will define S in multiple steps. Intuitively, S is defined by first pruning branches of T of small radius, and then *pumping* (in the theory of formal languages sense) long paths to make them even longer. In more detail, in Sect. 4.1 we will define the concept of a *skeleton tree*, where, starting from a tree T we define a tree T' where all subtrees of T having a height that is less than some threshold are removed. Then, in Sect. 4.2, we will prune T' even more, by removing all nodes of T' having degree strictly greater than 2, obtaining forest T''. This new forest will be a collection of paths. We will then split these paths in shorter paths and pump each of them. Intuitively, the pump procedure will replace the middle part of these paths by a repeated pattern that depends on the original content of the paths and on the parts previously removed when going from T to T'. Tree S will be obtained starting from the result of the pumping procedure, by bringing back the parts removed when going from T to T'. Also, throughout the definition of S we will keep track of a (partial) mapping between nodes of T and nodes of S.

Then, in Sect. 4.3 we will prove useful properties of S. One crucial property, shown in Lemma 3, will be that, if two nodes in T are far enough (that is, at $\omega(\sqrt{n})$ distance), then their corresponding nodes of S will be at much larger distance. In Sect. 4.4 we will use this property to show that we can execute $\mathcal{A}$ on S by inspecting only a neighbourhood of radius $O(\sqrt{n})$ of T. Notice that we will have conflicting requirements. On one hand, by pumping enough, the size of the graph increases to $N \gg n$, and $\mathcal{A}$ on S will be allowed to run for $t = o(N)$ rounds, that is, much more than the time allowed on T. This seems to give us the opposite effect of what we want, that is, we actually increased the running time instead of reducing it. On the other hand, we will prove that seeing at distance t on S requires to see only at distance $O(\sqrt{n})$ on T, hence we will effectively be able to run $\mathcal{A}$ within the required time bound. We will use the output given by $\mathcal{A}$ on S to obtain a partial solution for T, that is, only some nodes of T will fix their

Fig. 5 An example of a tree T and its skeleton T'; nodes removed from T in order to obtain T' are shown in grey. In this example, τ is 3
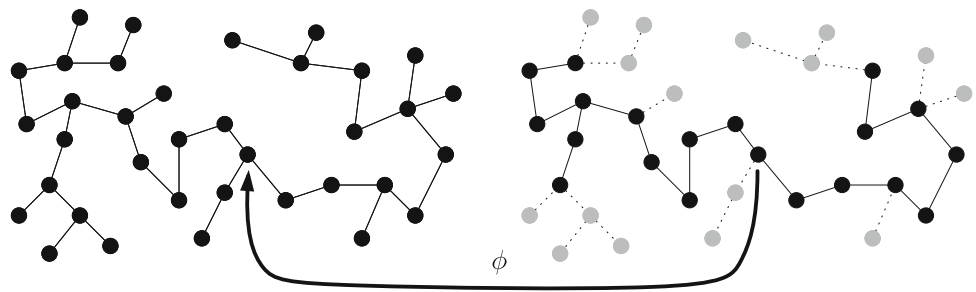
output using the same output of their corresponding node in S. There will be some nodes that remain unlabelled: those nodes that correspond to the pumped regions of S. Finally, in Sect. 4.5, we will show that it is possible to complete the unlabelled regions of T efficiently in a valid manner, heavily using techniques already presented in [9].

4.1 Skeleton tree

We first describe how, starting from a tree $T = (V, E)$, nodes can distributedly construct a virtual tree T', called the *skeleton* of T. Intuitively, T' is obtained by removing all subtrees of T having a height that is less than some threshold τ.

More formally, let $\tau = c\sqrt{n}$, for some constant c that will be fixed later. Each node v starts by gathering its τ-radius neighbourhood, Ball_v. Also, let d_v be the degree of node v in T. For all $v \in V$, we partition nodes of Ball_v (excluding v) in d_v components (one for each neighbour of v). Let us denote these components with $C_i(v)$, where $1 \le i \le d_v$. Each component $C_i(v)$ contains all nodes of Ball_v present in the subtree rooted at the i-th neighbour of v, excluding v.

Then, each node marks as Del all the components that have low depth and broadcasts this information. Informally, nodes build the skeleton tree by removing all the components that are marked as Del by at least one node. More precisely, each node v, for each $C_i(v)$, if $\mathrm{dist}(v, w) < \tau$ for all w in $V(C_i(v))$, marks all edges in $E(C_i(v)) \cup \{\{v, u\}\}$ as Del, where u is the i-th neighbour of v. Then, v broadcasts Ball_v and the edges marked as Del to all nodes at distance at most $\tau + 2c$. Finally, when a node v receives messages containing edges that have been marked with Del by some node, then also v internally marks as Del those edges.

Now we have all the ingredients to formally describe how we construct the skeleton tree. The skeleton tree $T' = (V', E')$ is defined in the following way. Intuitively, we keep only edges that have not been marked Del, and nodes with at least one remaining edge (i.e., nodes that have at least one incident edge not marked with Del). In particular,

$$E' = \left\{ e \in E(T) \mid e \text{ is not marked with Del} \right\},$$
$$V' = \left\{ u \in V \mid \text{there is a } w \in V \text{ s.t. } \{u, w\} \in E' \right\}.$$

Also, we want to keep track of the mapping from a node of T' to its original node in T; let $\phi \colon V(T') \to V(T)$ be such a mapping. Finally, we want to keep track of deleted subtrees, so let $\mathcal{T}_v$ be the subtree of T rooted at $v \in V'$ containing all nodes of $C_j(v)$, for all j such that $C_j(v)$ has been marked as Del. See Fig. 5 for an example.

4.2 Virtual tree

We now show how to distributedly construct a new virtual tree, starting from T', that satisfies some useful properties. The high level idea is the following. The new tree is obtained by *pumping* all paths contained in T' having length above some threshold. More precisely, by considering only degree-2 nodes of T' we obtain a set of paths. We split these paths in shorter paths of length l ($c \le l \le 2c$) by computing a $(c + 1, c)$ ruling set. Then, we pump these paths in order to obtain the final tree. Recall a (α, β) ruling set R of a graph G guarantees that nodes in R have distance at least α, while nodes outside R have at least one node in R at distance at most β. It can be distributedly computed in $O(\log^* n)$ rounds using standard colouring algorithms [17].

More formally, we start by splitting the tree in many paths of short length. Let T'' be the forest obtained by removing from T' each node v having $d_v^{T'} > 2$ (that is, the degree of v in T'). T'' is a collection $\mathcal{P}$ of disjoint paths. Let $\psi \colon V(T'') \to V(T')$ be the mapping from nodes of T'' to their corresponding node of T'. See Fig. 6 for an example.

We now want to split long paths of $\mathcal{P}$ in shorter paths. In order to achieve this, nodes of the same path can efficiently find a $(c + 1, c)$ ruling set in the path containing them. Nodes not in the ruling set form short paths of length l, such that $c \le l \le 2c$, except for some paths of $\mathcal{P}$ that were already too short, or subpaths at the two ends of a longer path (this can happen when a ruling set node happens to be very near to the endpoint of a path of $\mathcal{P}$). Let $\mathcal{Q}$ be the subset of the resulting paths having length l satisfying $c \le l \le 2c$. See Fig. 7 for an example.

In order to obtain the final tree, we will replace paths in $\mathcal{Q}$ with longer version of them. We will first describe a function, Replace, that can be used to replace a subgraph with a different one. Informally, given a graph G and a subgraph H

Fig. 6 An example of the tree T'' obtained from T'; nodes with degree greater than 2 (in blue) are removed from T' (color figure online)
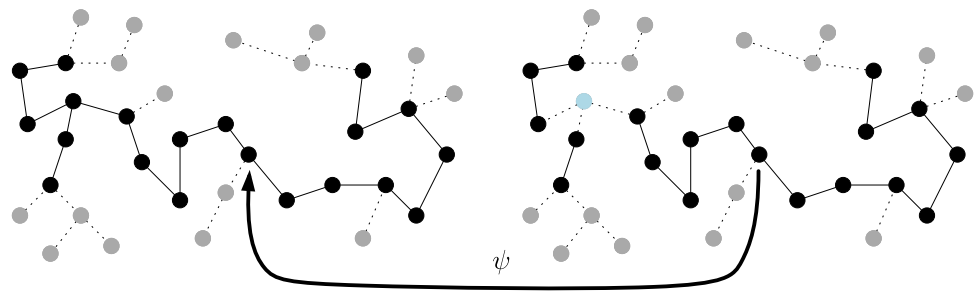
Fig. 7 Blue nodes break the long paths $\mathcal{P}$ of T'' shown on the left into short paths $\mathcal{Q}$ shown in black on the right; short paths (in the example, paths with length less then 4) are ignored (color figure online)
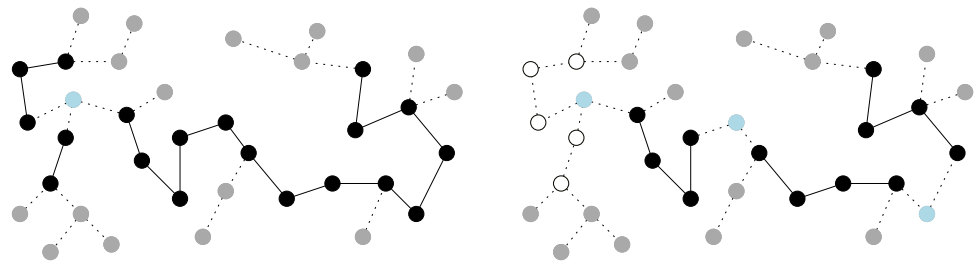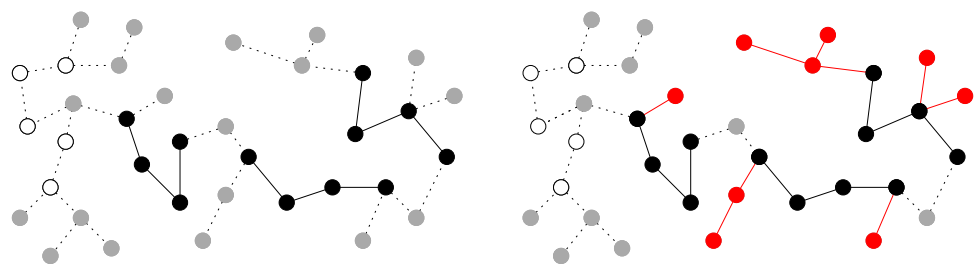
Fig. 8 An example of $\mathcal{Q}^T$, obtained by merging the path nodes (in black) with previously removed trees connected to them (in red) (color figure online)

connected to the other nodes of G via a set of nodes F called poles, and given another graph H', it replaces H with H'. This function is a simplified version of the function Replace presented in [9, Section 3.3].

Definition 1 (Replace) Let H be a subgraph of G, and let H' be an arbitrary graph. The poles of H are those vertices in $V(H)$ adjacent to some vertex in $V(G)\setminus V(H)$. Let $F = (v_1, \ldots, v_p)$ be a list of the poles of H, and let $F' = (v'_1, \ldots, v'_p)$ be a list of nodes contained in H' (called poles of H'). The graph $G' = $ Replace$(G, (H, F), (H', F'))$ is defined in the following way. Beginning with G, replace H with H', and replace any edge $\{u, v_i\}$, where $u \in V(G)\setminus V(H)$, with $\{u, v'_i\}$.

Informally, we will use the function Replace to substitute each path $Q \in \mathcal{Q}$ with a longer version of it, that satisfies some useful properties. In Sect. 4.5 we will have enough ingredients to be able to define a function, Pump, that is used to obtain these longer paths. This function will be defined in an analogous way of the function Pump presented in [9, Section 3.8]. For now, we just define some properties that this function must satisfy.

Definition 2 (*Properties of* Pump) Given a path $Q \in \mathcal{Q}$ of length l ($c \leq l \leq 2c$), consider the subgraph Q^T of T,

containing, for each $v \in V(Q)$, the tree $\mathcal{T}_{\chi(v)}$ (recall that $\mathcal{T}_v$ is the tree rooted at v containing nodes removed when pruning T, defined at the end of Sect. 4.1), where $\chi(v) = \phi(\psi(v)))$, that is, the path Q augmented with all the nodes deleted from the original tree that are connected to nodes of the path. Let v_1, v_2 be the endpoints of Q.

The function Pump(Q^T, B) produces a new tree P^T having two nodes, v'_1 and v'_2, satisfying that the path between v'_1 and v'_2 has length l', such that $cB \leq l' \leq c(B + 1)$. The new tree is obtained by replacing a subpath of Q, along with the deleted nodes connected to it, with many copies of the replaced part, concatenated one after the other. Pump satisfies that nodes $v'_1, v'_2 \in G'$, where $G' = $ Replace$(G, (Q^T, (v_1, v_2)), (P^T, (v'_1, v'_2)))$, have the same view as $v_1, v_2 \in G$ at distance $2r$ (where r is the LCL checkability radius). Note that, in the formal definition of Pump, we will set c as a function of r.

Let $\mathcal{Q}^T$ be the set containing all Q^T. See Fig. 8 for an example of $\mathcal{Q}^T$.

The final tree S is obtained from T by replacing each path $Q \in \mathcal{Q}$ in the following way. Replace each subgraph Q^T with $P^T = $ Pump(Q^T, B). Note that a node v cannot see the whole set $\mathcal{Q}$, but just all the paths $Q \in \mathcal{Q}$ that end at distance at most $\tau + 2c$ from v. Thus each node locally computes just a part of S, that is enough for our purpose.

Fig. 9 Tree S (on the right) is obtained by pumping the black paths
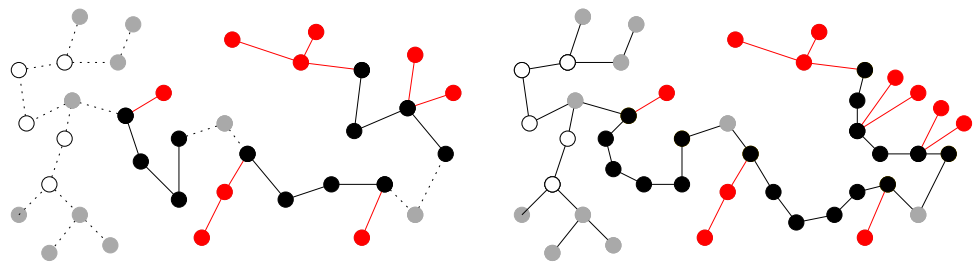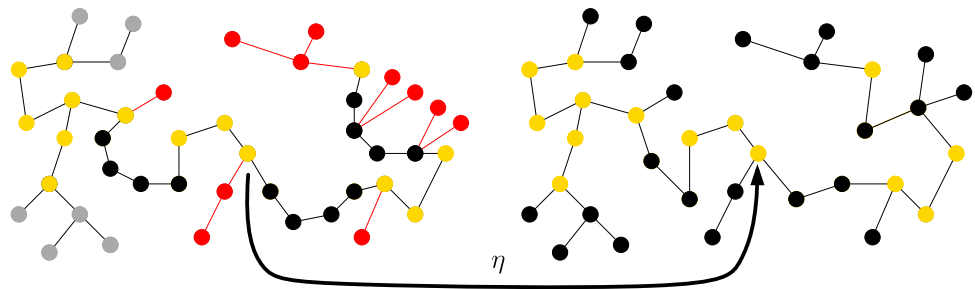

Fig. 10 Nodes in yellow on the left are the ones in S_o, while the yellow ones on the right are nodes in T_o. Note that, for the sake of simplicity, we consider $2r = 1$ (color figure online)

We call the subgraph of Q^T induced by the nodes of Q the *main path* of Q^T, and we define the main path of P^T in an analogous way. See Fig. 9 for an example.

Finally, we want to keep track of the *real* nodes of S, that will be nodes that have not been removed when creating the skeleton tree T' and are also not part of the pumped regions. Nodes of S are divided in two parts, S_o and S_p. The set S_o contains all nodes of T' that are not contained in any Q^T, and all nodes that are at distance at most $2r$ from nodes not contained in any Q^T, while $S_p = V(S) \backslash S_o$. Let η be a mapping from real nodes of the virtual graph (S_o) to their corresponding node of T (this is well defined, by the properties of Pump), and let $T_o = \{\eta(v) \mid v \in S_o\}$ (note that also η^{-1} is well defined for nodes in T_o). Informally, T_o is the subset of nodes of T that are far enough from pumped regions of S, and have not been removed while creating T'. Note that we use the function η to distinguish between nodes of S and nodes of T, but η is actually the identity function between a subset of shared nodes. This concludes the definition of S, as a function of the original tree T, and two parameters, B and c. Let Virt be the function that maps T to S, that is, $S = \text{Virt}(T, B, c)$. See Fig. 10 for an example.

4.3 Properties of the virtual tree

We will now prove three properties about the virtual graph S. The first one provides an upper bound on the number of nodes of S, as a function of the number of nodes of T. This will be useful when executing $\mathcal{A}$ on S. In that case, we will lie to $\mathcal{A}$ about the size of the graph, by telling to the algorithm that there are $N = c(B+1)n$ nodes. This lemma will guarantee us that the algorithm cannot see more than N nodes and notice an inconsistency.

Lemma 1 *The tree S has at most $N = c(B + 1)n$ nodes, where $n = |V(T)|$, and $S = \text{Virt}(T, B, c)$.*

Proof S is obtained by pumping T. The main path of the subtree obtained by pumping some $Q^T \in \mathcal{Q}^T$ has length at most $c(B + 1)$. This implies that each node of the main path of Q^T is copied at most $c(B + 1)$ times. Also, a deleted tree $\mathcal{T}_v$ rooted at some path node v is not connected to more than one path node. Thus, *all* nodes of T are copied at most $c(B + 1)$ times. □

The following lemma bounds the size of T' compared to the size of T''. This bound will be useful in the proof of Lemma 3. Notice that, this is the exact point in which our approach stops working for time complexities of $O(\sqrt{n})$ rounds. This is exactly what we expect, since we know that there are LCL problems on trees having complexity $\Theta(\sqrt{n})$ [9].

Lemma 2 *For any path $P = (x_1, \ldots, x_k)$ of length $k \geq c\sqrt{n}$ that is a subgraph of T', at most $\frac{\sqrt{n}}{c}$ nodes in $V(P)$ have degree greater than 2.*

Proof If a node $x_j \in P$ has $d_v^{T'} > 2$, it means that it has at least one neighbour $z \notin \{x_{j-1}, x_{j+1}\}$ in T' such that there exists a node w satisfying $\text{dist}(x_j, w) \geq \tau$ such that the shortest path connecting x_j and w contains z. Thus, for each node in P with $d_v^{T'} > 2$, we have at least other τ nodes not in P. If at least $\frac{\sqrt{n}}{c} + 1$ nodes of P have degree greater than 2, we would obtain a total of $(\frac{\sqrt{n}}{c} + 1) \cdot \tau > n$ nodes, a contradiction. □

The following lemma compares distances in T with distances in S, and states that if two nodes are far enough, that is, at $\omega(\sqrt{n})$ distance in T, then we can increase the distance

of their corresponding nodes in S by an arbitrary amount. This is what will allow us to speedup $\mathcal{A}$ to $O(\sqrt{n})$.

Lemma 3 *There exists some constant c such that, if nodes u, v of T_o are at distance at least $c\sqrt{n}$ in T, then their corresponding nodes $\eta^{-1}(u)$ and $\eta^{-1}(v)$ are at distance at least $cB\sqrt{n}/3$ in S.*

Proof Consider a node u at distance at least τ from v in T. There must exist a path P in T' connecting $\phi^{-1}(u)$ and $\phi^{-1}(v)$. By Lemma 2, at most $\frac{\sqrt{n}}{c}$ nodes in P have degree greater than 2, call the set of these nodes X. We can bound the number of nodes of P that are not part of paths that will be pumped in the following way:

– At most $\frac{c\sqrt{n}+1}{c+1} + \frac{\sqrt{n}}{c} + 1$ nodes can be part of the ruling set. To see this, order the nodes of P from left to right in one of the two canonical ways. The first summand bounds all the ruling set nodes whose right-hand short path is of length at least c, the second one bounds the ruling set nodes whose right-hand short path ends in a node $x \in X$, and the last one considers the path that ends in $\phi^{-1}(u)$ or $\phi^{-1}(v)$.
– At most $\frac{\sqrt{n}}{c}(1 + 2(c - 1))$ nodes are either in X or in short paths of length at most $c - 1$ on the sides of a node in X.
– At most $2(c - 1)$ nodes are between $\phi^{-1}(u)$ (or $\phi^{-1}(v)$) and a ruling set node.

While pumping the graph, in the worst case we replace paths of length $2c$ with paths of length cB, thus

$$\text{dist}(\phi^{-1}(u), \phi^{-1}(v))$$
$$\geq \left(c\sqrt{n} + 1 - \left(\frac{c\sqrt{n}+1}{c+1} + \frac{\sqrt{n}}{c} + 1 \right.\right.$$
$$\left.\left. + \frac{\sqrt{n}}{c}\big(1 + 2(c - 1)\big) + 2(c - 1) \right) \right) \cdot \frac{cB}{2c} - 1,$$

which is greater than $cB\sqrt{n}/3$ for c and n greater than a large enough constant. $\square$

4.4 Solving the problem faster

We now show how to speed up the algorithm $\mathcal{A}$ and obtain an algorithm running in $O(\sqrt{n})$. First, note that if the diameter of the original graph is $O(\sqrt{n})$, every node sees the whole graph in $O(\sqrt{n})$ rounds, and the problem is trivially solvable by brute force. Thus, in the following we assume that the diameter of the graph is $\omega(\sqrt{n})$. This also guarantees that T_o is not empty.

Informally, nodes can distributedly construct the virtual tree S in $O(\sqrt{n})$ rounds, and safely execute the original algorithm on it. Intuitively, even if a node v sees just a part of S,

we need to guarantee that this part has large enough radius, such that the original algorithm cannot see outside the subgraph of S constructed by v (otherwise v would not be able to simulate the execution of $\mathcal{A}$ on S).

More precisely, all nodes do the following. First, they distributedly construct S, in $O(\sqrt{n})$ rounds. This increases the number of nodes, and requires nodes to assign new unique IDs to nodes that do not exist in the original graph (that is, nodes in the pumped regions). This new ID assignment can be computed in a standard manner as a function of the two IDs of the endpoints of the pumped paths. Then, each node v in T_o (nodes for which $\eta^{-1}(v)$ is defined), simulates the execution of $\mathcal{A}$ on node $\eta^{-1}(v)$ of S, by telling $\mathcal{A}$ that there are $N = c(B + 1)n$ nodes. Then, each node v in T_o outputs the same output assigned by $\mathcal{A}$ to node $\eta^{-1}(v)$ in S. Also, each node v in T_o fixes the output for all nodes in $\mathcal{T}_v$ (η can be defined also for them, v sees all of them, and the view of these nodes is contained in the view of v, thus it can simulate $\mathcal{A}$ in S for all of them). Let Λ be the set of nodes that already fixed an output, that is, $\Lambda = \{\{u\} \cup V(\mathcal{T}_u) \mid u \in T_o\}$. Intuitively Λ contains all the real nodes of S (nodes with a corresponding node in T), including nodes removed when computing the skeleton tree, and leaves out only nodes that correspond to pumped regions. Finally, nodes in $V(T)\backslash\Lambda$ find a valid output via brute force.

We need to prove two properties, the first shows that a node can safely execute $\mathcal{A}$ on the subgraph of S that it knows, while the second shows that it is always possible to find a valid output for nodes in $V(T)\backslash\Lambda$ after having fixed outputs for nodes in Λ.

Let us choose a B satisfying $\tau_{\text{orig}}(N) \leq cB\sqrt{n}/3$, where $\tau_{\text{orig}}(N)$ is the running time of $\mathcal{A}$. Note that B can be an arbitrarily large function of n. Such a B exists for all $\tau_{\text{orig}}(x) = o(x)$. We prove the following lemma.

Lemma 4 *For nodes in T_o, it is possible to execute $\mathcal{A}$ on S by just knowing the neighbourhood of radius $2c\sqrt{n}$ in T.*

Proof First, note that by Lemma 1, the number of nodes of the virtual graph, $|V(S)|$, is always at most N, thus, it is not possible that a node of S sees a number of nodes that is more than the number claimed when simulating the algorithm.

Second, since B satisfies $\tau_{\text{orig}}(N) \leq cB\sqrt{n}/3$, and since, by Lemma 3 and the bound of $c\sqrt{n}$ on the depth of each deleted tree $\mathcal{T}_u$, the nodes outside a $2c\sqrt{n}$ ball of nodes in T_o are at distance at least $cB\sqrt{n}/3$ in S, the running time of $\mathcal{A}$ is less than the radius of the subtree of S rooted at a node v that v distributedly computed and is aware of. This second part also implies that nodes in T_o do not see the whole graph, thus they cannot notice that the value of N is not the real size of the graph. $\square$

4.5 Filling gaps by brute force

In this last part, we show that, by starting from a tree T in which nodes of Λ have already fixed an output, we can find a valid output for all the other nodes of the graph, in constant time. For this purpose, we adapt some definitions presented in [9], where it is shown that, by starting from a partially labelled graph, if we replace a subgraph with a different subgraph of the same *type*, then the labelling of the original graph can be completed if and only if the labelling of the new graph can be completed. In our case the subgraphs that we replace are not labelled, and the following definitions handle exactly this case. In the following, unless stated otherwise, we use the term labelling to refer to an *output* labelling.

We start by defining an equivalence relation $\overset{*}{\sim}$ between two pairs (H, F) and (H', F') composed of a graph and its poles. Intuitively, this equivalence relation says that equivalent H and H' should be isomorphic near the poles, and that if we fix some output near the poles of one graph, if we copy that output on the other graph (on the isomorphic part), and if that output is completable on the remaining nodes of the first graph, then it should be completable also on the other graph. A partial labelling (a partial function from nodes to labels) is called *extendible* if it is possible to assign a label to unlabelled nodes such that it is locally consistent for every node, that is, the labelling satisfies the constraints of the given LCL problem at every node. This is a simplified version of the equivalence relation $\overset{*}{\sim}$ presented in [9, Section 3.5].

Definition 3 (*The equivalence relation $\overset{*}{\sim}$*) Given a graph H and its poles F, define $\xi(H, F) = (D_1, D_2, D_3)$ to be a tripartition of $V(H)$ where

$$D_1 = \bigcup_{v \in F} N^{r-1}(v),$$
$$D_2 = \bigcup_{v \in D_1} N^r(v) \backslash D_1,$$
$$D_3 = V(H) - (D_1 \cup D_2).$$

Let Q and Q' be the subgraphs of H and H' induced by the vertices in $D_1 \cup D_2$ and $D'_1 \cup D'_2$ respectively.

The equivalence holds, i.e., $(H, F) \overset{*}{\sim} (H', F')$, if and only if there is a 1–1 correspondence $\phi \colon (D_1 \cup D_2) \to (D'_1 \cup D'_2)$ satisfying:

- Q and Q' are isomorphic under ϕ, preserving the input labels of the LCL problem (if any), and preserving the order of the poles.
- Let $\mathcal{L}_*$ be any assignment of output labels to vertices in $D_1 \cup D_2$, and let $\mathcal{L}'_*$ be the corresponding labelling of $D'_1 \cup D'_2$ under ϕ. Then $\mathcal{L}_*$ is extendible to $V(H)$ if and only if $\mathcal{L}'_*$ is extendible to $V(H')$.

In [9], it is proved that this equivalence relation is preserved after replacing equivalent subgraphs, and that, if the number of poles is constant, there is a constant number of equivalence classes.

Also, in [9, Section 3.6] the following lemma is proved. Informally, it shows that if we have a valid labelling for a graph, and we replace a subgraph with another equivalent to it, it is enough to change the labelling of the new subgraph in order to obtain a valid labelling for the whole new graph. Also, the labelling near the borders is preserved. A labelling is locally consistent for node v if the LCL verifier running on node v accepts that labelling.

Lemma 5 *Let $G' = \mathsf{Replace}(G, (H, F), (H', F'))$. Suppose $(H, F) \overset{*}{\sim} (H', F')$. Let $D_0 = V(G) \backslash V(H)$. Let $\mathcal{L}_\diamond$ be a complete labelling of G that is locally consistent for all vertices in $D_2 \cup D_3$. Then there exists a complete labelling $\mathcal{L}'_\diamond$ satisfying the following:*

- *$\mathcal{L}_\diamond = \mathcal{L}'_\diamond$ for all $v \in D_0 \cup D_1 \cup D_2$ and their corresponding vertices in $D'_0 \cup D'_1 \cup D'_2$. Also, if $\mathcal{L}_\diamond$ is locally consistent for a node v, then $\mathcal{L}'_\diamond$ is locally consistent for $\phi(v)$.*
- *$\mathcal{L}'_\diamond$ is locally consistent for all nodes in $D'_2 \cup D'_3$.*

We now adapt the definition of the function Pump presented in [9] for our purposes. Intuitively, as previously explained, starting from our tree T we replace all subgraphs $Q^T \in \mathcal{Q}^T$ with a pumped version of Q^T. Each Q^T is composed of a main path in which, for each node, there is a subtree of height $O(\sqrt{n})$. Note that Q^T is connected to the rest of T on the two endpoints of the main path, thus it has two poles, that implies, as previously discussed, that the number of equivalence classes under $\overset{*}{\sim}$ is constant. This class is also computable by a node, since it considers only the subtrees Q^T that are contained in its ball. Also, we can see Q^T as a sequence of $\mathcal{T}_v$, and the type of a Q^T can be computed, as in [9], by reading one "character" (class of $\mathcal{T}_v$) at a time. Finally, we can see the sequence as a string that, if it is long enough, we can pump in order to obtain a longer string of the same type. More formally, consider a tree $Q^T \in \mathcal{Q}^T$. We can see Q^T as a path of length k, where each node i is the root of a tree $\mathcal{T}_i$ $(1 \le i \le k)$. Let $(\mathcal{T}_i)_{i \in [k]}$ denote this path. Let $\mathsf{Class}(\mathcal{T}_j)$ be the equivalence class of the tree $\mathcal{T}_j$ considering j as the unique pole, and let $\mathsf{Type}(H)$ be the equivalence class of the path H considering its endpoints as poles.

The following lemma says that nodes can compute the type of the deleted trees rooted on nodes contained in their balls.

Lemma 6 *Each node u can determine the type of $\mathcal{T}_v$ for all $v \in \mathsf{Ball}_u$.*

Proof When nodes compute the skeleton tree T', they broadcast all their balls to the nodes inside their balls. Since a tree $\mathcal{T}_v$ has height $O(\sqrt{n})$, it is fully contained in the ball of v, thus all the nodes in the ball of v see the whole tree $\mathcal{T}_v$, and can determine its type (it depends only on the structure of $\mathcal{T}_v$ and the inputs of the nodes in this tree). □

The following is a crucial lemma proved in [9, Section 3.8].

Lemma 7 Let $H = (\mathcal{T}_i)_{i \in [k]}$ and $H' = (\mathcal{T}_i)_{i \in [k+1]}$ be identical to H in its first k trees. Then $\mathrm{Type}(H')$ is a function of $\mathrm{Type}(H)$ and $\mathrm{Class}(\mathcal{T}_{k+1})$.

As shown in [9], Lemma 7 allows us to bring classic automata theory into play. By Lemma 6, nodes can know the type of each $\mathcal{T}_i$ contained in a path that they want to pump. Consider a path $H = (\mathcal{T}_i)_{i \in [k]}$, and the sequence $C = (c_1, \ldots, c_k)$, where c_i is $\mathrm{Class}(\mathcal{T}_i)$. A finite automaton can determine the type of H by reading one character of C at a time. The number of states in this automaton is constant, let ℓ_{pump} be such a constant. The following lemma holds [9, Lemma 7]).

Lemma 8 Let $H = (\mathcal{T}_i)_{i \in [k]}$, with $k \geq \ell_{\mathsf{pump}}$. H can be decomposed into three subpaths $H = x \circ y \circ z$ such that:

- $|xy| \leq \ell_{\mathsf{pump}}$,
- $|y| \geq 1$,
- $\mathrm{Type}(x \circ y^j \circ z) = \mathrm{Type}(H)$ for each non-negative j.

We finally define the function Pump, that, given a tree Q^T having a main path of short length, produces a new tree P^T having a main path that is arbitrary longer, such that their types are equivalent.

Definition 4 (Pump) Let $Q^T \in \mathcal{Q}^T$, and fix $c = \ell_{\mathsf{pump}} + 4r$. We have that the main path of Q^T has length at least $\ell_{\mathsf{pump}} + 4r$. Let us split the main path of Q^T in three subpaths p_l, p_c, p_r, two of length $2r$ near the poles (p_l and p_r), and one of length at least ℓ_{pump} containing the remaining nodes (p_c). $\mathsf{Pump}(Q^T, B)$ produces a tree P^T such that $\mathrm{Type}(Q^T) = \mathrm{Type}(P^T)$ and the main path of P^T has length l' satisfying $cB \leq l' \leq c(B+1)$. This is obtained by pumping the subpath p_c. By Lemma 8 such a function exists. Since the paths p_l and p_r are preserved during the pump, the isomorphism near the poles is preserved.

We now prove that the partial labelling produced by the algorithm previously described can be completed consistently. Consider the tripartition described in Definition 3. Let $\mathcal{R}$ be the union of all the replaced subgraphs, and let D_1, D_2, D_3 be a tripartition of it as defined in Definition 3. By definition of T_o, Λ corresponds to nodes in $D_0 \cup D_1 \cup D_2$.

First, notice that a node in $\mathcal{R}$ sees all the nodes in the regions D_1 and D_2 of the replaced subgraph where it is located, thus it has enough information needed to find a valid output via brute force.

Second, by Lemma 5, in order to show that the partial labelling can be completed consistently, it is enough to show that each replaced Q^T is in the same equivalence class as $\mathsf{Pump}(Q^T, B)$, which is true by the definition of Pump.

References

1. Balliu, A., Brandt, S., Olivetti, D., Suomela, J.: Almost global problems in the LOCAL model. In: Proceedings 32nd International Symposium on Distributed Computing (DISC 2018), Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz–Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.DISC.2018.9

2. Balliu, A., Hirvonen, J., Korhonen, J.H., Lempiäinen, T., Olivetti, D., Suomela, J.: New classes of distributed time complexity. In: Proceedings 50th ACM Symposium on Theory of Computing (STOC 2018), pp. 1307–1318. ACM Press (2018). https://doi.org/10.1145/3188745.3188860

3. Barenboim, L.: Deterministic (Δ+1)-coloring in sublinear (in Δ) time in static, dynamic, and faulty networks. J. ACM **63**(5), 1–22 (2016). https://doi.org/10.1145/2979675

4. Barenboim, L., Elkin, M., Kuhn, F.: Distributed (Δ+1)-coloring in linear (in Δ) time. SIAM J. Comput. **43**(1), 72–95 (2014). https://doi.org/10.1137/12088848X

5. Brandt, S., Fischer, O., Hirvonen, J., Keller, B., Lempiäinen, T., Rybicki, J., Suomela, J., Uitto, J.: A lower bound for the distributed Lovász local lemma. In: Proceedings 48th ACM Symposium on Theory of Computing (STOC 2016), pp. 479–488. ACM Press (2016). https://doi.org/10.1145/2897518.2897570

6. Brandt, S., Hirvonen, J., Korhonen, J.H., Lempiäinen, T., Östergård, P.R.J., Purcell, C., Rybicki, J., Suomela, J., Uznański, P.: LCL problems on grids. In: Proceedings 36th ACM Symposium on Principles of Distributed Computing (PODC 2017), pp. 101–110. ACM Press (2017). https://doi.org/10.1145/3087801.3087833

7. Chang, Y.J., He, Q., Li, W., Pettie, S., Uitto, J.: Distributed edge coloring and a special case of the constructive Lovász local lemma. ACM Trans. Algorithms **16**(1), 8:1–8:51 (2020). https://doi.org/10.1145/3365004

8. Chang, Y.J., Kopelowitz, T., Pettie, S.: An exponential separation between randomized and deterministic complexity in the LOCAL

model. SIAM J. Comput. **48**(1), 122–143 (2019). https://doi.org/10.1137/17M1117537

9. Chang, Y.J., Pettie, S.: A time hierarchy theorem for the LOCAL model. SIAM J. Comput. **48**(1), 33–69 (2019). https://doi.org/10.1137/17M1157957

10. Cole, R., Vishkin, U.: Deterministic coin tossing with applications to optimal parallel list ranking. Inf. Control **70**(1), 32–53 (1986). https://doi.org/10.1016/S0019-9958(86)80023-7

11. Fraigniaud, P., Heinrich, M., Kosowski, A.: Local conflict coloring. In: Proceedings 57th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2016), pp. 625–634. IEEE (2016). https://doi.org/10.1109/FOCS.2016.73

12. Ghaffari, M., Harris, D.G., Kuhn, F.: On derandomizing local distributed algorithms. In: Proceedings 59th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2018), pp. 662–673. IEEE (2018). https://doi.org/10.1109/FOCS.2018.00069

13. Ghaffari, M., Su, H.H.: Distributed Degree Splitting, Edge Coloring, and Orientations. In: Proceedings 28th ACM-SIAM Symposium on Discrete Algorithms (SODA 2017), pp. 2505–2523. Society for Industrial and Applied Mathematics (2017). https://doi.org/10.1137/1.9781611974782.166

14. Göös, M., Suomela, J.: Locally checkable proofs in distributed computing. Theory Comput. **12**, 1–33 (2016). https://doi.org/10.4086/toc.2016.v012a019

15. Hartmanis, J., Stearns, R.E.: On the computational complexity of algorithms. Trans. Am. Math. Soc. **117**(117), 285–285 (1965). https://doi.org/10.1090/S0002-9947-1965-0170805-7

16. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Boston (1979)

17. Linial, N.: Locality in distributed graph algorithms. SIAM J. Comput. **21**(1), 193–201 (1992). https://doi.org/10.1137/0221015

18. Naor, M., Stockmeyer, L.: What can be computed locally? SIAM J. Comput. **24**(6), 1259–1277 (1995). https://doi.org/10.1137/S0097539793254571

19. Panconesi, A., Rizzi, R.: Some simple distributed algorithms for sparse networks. Distributed Comput. **14**(2), 97–100 (2001). https://doi.org/10.1007/PL00008932

20. Panconesi, A., Srinivasan, A.: The local nature of Δ-coloring and its algorithmic applications. Combinatorica **15**(2), 255–280 (1995). https://doi.org/10.1007/BF01200759

21. Peleg, D.: Distributed Computing: A Locality-Sensitive Approach. Society for Industrial and Applied Mathematics, Philadelphia (2000). https://doi.org/10.1137/1.9780898719772