Balliu, Alkida; Brandt, Sebastian; Olivetti, Dennis; Suomela, Jukka

Almost global problems in the LOCAL model

Published in:
32nd International Symposium on Distributed Computing (DISC 2018)

DOI:
10.4230/LIPIcs.DISC.2018.9

Published: 01/01/2018

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

Almost Global Problems in the LOCAL Model

Alkida Balliu
Aalto University, Finland
alkida.balliu@aalto.fi

Sebastian Brandt
ETH Zürich, Switzerland
brandts@ethz.ch

Dennis Olivetti
Aalto University, Finland
dennis.olivetti@aalto.fi

Jukka Suomela
Aalto University, Finland
jukka.suomela@aalto.fi

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 $\Theta(\log^* n)$ 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 problems with complexities of the form $\Theta(n^\alpha)$ for any rational $0 < \alpha \leq 1/2$, 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.

2012 ACM Subject Classification Theory of computation → Distributed computing models, Theory of computation → Complexity classes

Keywords and phrases Distributed complexity theory, locally checkable labellings, LOCAL model

Digital Object Identifier 10.4230/LIPIcs.DISC.2018.9

Related Version The full version is available at `https://arxiv.org/pdf/1805.04776.pdf`.

Funding This work was supported in part by the Academy of Finland, Grant 285721.

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.

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 Section 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 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 the distributed time complexity of solving a given LCL problem with *deterministic* distributed algorithms is.

Distributed structural complexity. In the classical (centralised, sequential) complexity theory one of the cornerstones is the *time hierarchy theorem* [12]. 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 $o(\log n)$ [7].

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 $\operatorname{polylog} n$ time [17]. Hence 4-colouring in 2-dimensional grids can be also solved in $\operatorname{polylog} n$ time. But we also know that in 2-dimensional grids there is a gap in distributed time complexities between $\omega(\log^* n)$ and $o(\sqrt{n})$ [5], 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.

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 $\Theta(n^{1/k})$ [8].
- In bounded-degree graphs, for each rational number $0 < \alpha \leq 1/2$ there is an LCL problem with time complexity $\Theta(n^\alpha)$ [1].

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 $1/2 < \alpha < 1$.

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.

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.

In the positive result, the key idea 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 [1] 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 [8] 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.

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 [14, 18]. 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. Hence, after t rounds in the LOCAL model, each node has knowledge about the network up to distance t from him. The time complexity of an algorithm running in the LOCAL model is determined by this radius-t that each node needs to explore in order to solve a given problem. It is easy to see that, in this setting, every problem can be solved in *diameter* time.

2.2 Locally checkable labellings

Locally checkable labelling problems (LCLs) were introduced in the seminal work of Naor and Stockmeyer [15]. These problems are defined on bounded degree graphs, so let $\mathcal{F}$ be the family of such graphs. Also, let Σ_{in} and Σ_{out} be respectively input and output label alphabets. Each node v of a graph $G \in \mathcal{F}$ has an input $i(v) \in \Sigma_{\mathsf{in}}$, and must produce an output $o(v) \in \Sigma_{\mathsf{out}}$. The output that each node must produce depends on the constraints defined with the LCL problem. Hence, let C be the set of legal configurations. A problem Π is an LCL problem if

- Σ_{in} and Σ_{out} are of constant size;
- there exists an algorithm $\mathcal{A}$ able to check the validity of a solution in constant time in the LOCAL model.

Hence, if the solution produced by the nodes is in the set C of valid configurations, then, by just looking at its local neighbourhood, each node must output 'accept', otherwise, at least one node must output 'reject'. An example of an LCL problem is vertex colouring, where we have a constant size palette of colours; nodes can easily check in 1 round whether the produced colouring is valid or not.

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 [9, 14]. Chang, Kopelowitz, and Pettie [7] 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.

Oriented grids. Brandt et al. [5] 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. However, while it is decidable whether a given LCL on cycles can be solved in t-rounds in the LOCAL model [5, 15], it is not the case for oriented grids [5].

Trees. Although well studied, LCLs on trees are not fully understood yet. Chang and Pettie [8] 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 $\omega(n^{1/(k+1)})$.

General graphs. Another important direction of research is understanding LCLs on general (bounded-degree) graphs. Using the techniques presented by Naor and Stockmeyer [15], 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 $\Theta(\log^* n)$ [2, 3, 10, 16] and $\Theta(\log n)$ [4, 7, 11]. On the other hand, Chang et al. [7] 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 [4, 6], for which there are algorithms solving them in $O(\text{polylog}\, n)$ rounds [17]. 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 [1] 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 the existence of LCL problems having complexities in the spectrum between $\omega(\sqrt{n})$ and $o(n)$. We first give the definition of a standard model of computation, that is Linear Bounded Automata, and we then show that it is possible to encode the execution of an LBA as a locally checkable labelling. We then define an LCL problem where interesting instances are those in which one encodes the execution of a specific LBA in a multidimensional grid. Depending on the number of dimensions of the grid, and on the running time of the LBA, we obtain different time complexities.

3.1 Linear bounded automata

A Linear Bounded Automaton (LBA) M_B consists of a Turing machine with a tape of bounded size B, able to recognize the boundaries of the tape [13, 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 (of size B) 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, the transition function δ is going to be described 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;
     - '$\leftarrow$' indicates that the head moves to the previous cell;
     - '$-$' means the head does not move.

If δ is not defined on the current state and tape content, the machine terminates. The *growth* of an LBA M_B, denoted with $g(M_B)$, is defined as the running time of M_B. For example, it is easy to design a machine M that implements a binary counter, counting from all-0 to all-1, and this gives a growth of $g(M_B) = \Theta(2^B)$.

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 achieve a growth

of $g(M_B) = \Theta(B^k)$ 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).

3.2 Grid structure

Each LCL problem we will construct in Section 3.4 is designed in a way that ensures that the hardest input graphs for the LCL problem, i.e., the graphs providing the lower bound instances for the claimed time complexity, have a (multidimensional) grid structure. In this section, we introduce a class of graphs with this structure.

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.

3.2.1 Grid labels

In addition to the graph structure, we add *constant-size* labels to each grid graph. 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. 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.

The labelling of the edges here is just a matter of convenience. We could equally well assign the labels to nodes instead of edges, satisfying the formal criteria of an LCL problem (and, for that matter, combine all input labels, and later output labels, of a node into a single input, resp. output, label). Furthermore, we could also equally well encode the labels in the graph structure. Hence all new time complexities presented in Section 3.4 can also be achieved by LCL problems without input labels.

In the full version of this paper we prove that, assuming that the considered graph contains a node not having any edge labelled with Prev_j, for all dimensions j, then nodes can locally check if they are in a valid grid graph.

3.2.2 Unbalanced grid graphs

In Section 3.2.1, 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 any constructed LCL, a locally checkable *proof* (of a certain well-specified kind) certifying that the input graph is an unbalanced grid graph constitutes a valid (global) output.

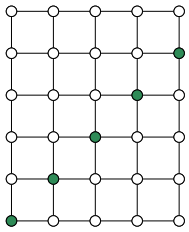

Figure 1 An example of an unbalanced grid with 2 dimensions; nodes in green are labelled with Unbalanced, while white nodes are labelled with Exempt.

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

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

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 it has exactly two "diagonal neighbours" and that their positions relative to v are consistent with the above output specification. Node v also may have only one diagonal neighbour, but only if it has no incident edge labelled Prev_j, or if it has an incident edge labelled Next_j for all $2 \leq j \leq i$, but no incident edge labelled Next_1. The latter condition ensures that the described diagonal chain of labels terminates at the end of dimension 1, but not at the end of any other dimension, thereby guaranteeing that grid graphs that are not unbalanced do not allow the output labelling specified above. Finally, the unique node without any incident edge labelled Prev_j checks that it is labelled Unbalanced, in order to prevent the possibility that each node simply outputs Exempt. We refer to Figure 1 for an example of an unbalanced 2-dimensional grid and its labelling.

3.3 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. Essentially, the execution of the LBA 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, we introduce the notion of an LBA *encoding graph* in the following.

3.3.1 Labels

Let M_B be an LBA, where B denotes the size of the tape. Let $S_\ell = (s_\ell, h_\ell, t_\ell)$ be the whole state of M_B 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 LBA *encoding graph* if there exists a dimension $2 \leq k \leq i$ satisfying the following.

- $d_k + 1$ is equal to B.
- For all $0 \leq x \leq \min\{g(M_B), d_1\}$ and all $0 \leq y \leq B - 1$, it holds that:
  - Node $(x, y)_k$ is labelled with $\mathsf{Tape}(t_x[y])$.
  - Node $(x, y)_k$ is labelled with $\mathsf{State}(s_x)$.
  - Node $(x, h_x)_k$ is labelled with Head.
  - Node $(x, y)_k$ is labelled with $\mathsf{Dimension}(k)$.
- All other nodes are labelled with Exempt.

Intuitively, the 2-dimensional surface expanding in dimensions 1 and k (having all the other coordinates equal to 0), encodes the execution of the LBA. The described labelling is locally checkable, see the full version of this paper for details.

3.4 LCL construction

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

- produce a valid LBA encoding, 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).

3.4.1 LCL Problem Π_M

Denote by $\mathcal{L}$ the set of output labels used for producing an LBA encoding graph. Formally, we specify the LCL problem Π_M as follows. The input label set for Π_M is the set of labels used in the grid labelling. The possible output labels are the following:

1. the labels from $\mathcal{L}$;
2. an *unbalanced label*, Unbalanced;
3. an *exempt label*, Exempt;
4. an *error label* Error;
5. *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.

Note that the separate mention of Exempt in this list is not strictly necessary since Exempt is contained in $\mathcal{L}$, but we want to recall the fact that Exempt can be used in both a proof of unbalance and an LBA encoding.

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 neighborhood*, 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

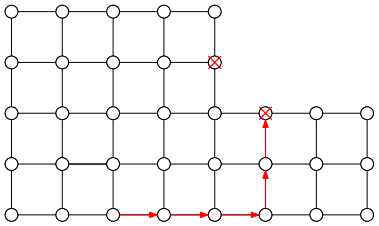

Figure 2 An example of an error pointer chain (shown in red). Nodes that are marked with a red cross are those who actually see an error in the grid structure. The output of only some of the depicted nodes is shown.

wraps around in some way, while type 1 pointers are for nodes that can actually see an error. 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), except in the case that the grid structure wraps around "nicely" (e.g., along one dimension). This exceptional case is the only scenario where, deviating from the above, an error pointer chain does not necessarily end in an error, but instead may form 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 Figure 2 for an example of an error pointer chain.

An output labelling for problem Π_M 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 $\mathcal{L} \setminus \{\mathsf{Exempt}\}$.
2. Each node at which the input labelling does not satisfy the local grid graph constraints given in Section 3.2.1 outputs Error. All other nodes do not output Error.
3. If a node v outputs Exempt, then v has at least one incident edge e with input label $L_v(e) \in \{\mathsf{Prev}_1, \ldots, \mathsf{Prev}_i\}$.
4. If the output labels of a node v are contained in $\mathcal{L} \setminus \{\mathsf{Exempt}\}$, then either there is a node in v's 2-radius neighbourhood that outputs an error pointer, or the output labels of all nodes in v's 2-radius neighbourhood are contained in $\mathcal{L}$. Moreover, in the latter case v's 2-radius neighbourhood has a valid grid structure and the local constraints of an LBA encoding graph, given in Section 3.1, are satisfied at v.
5. If the output of a node v is Unbalanced, then either there is a node in v's i-radius neighbourhood that outputs an error pointer, or the output labels of all nodes in v's i-radius neighbourhood are contained in $\{\mathsf{Unbalanced}, \mathsf{Exempt}\}$. 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 Section 3.2.2, 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 is a valid LCL problem.

3.4.2 Time complexity

Let B be the smallest positive integer satisfying $n \leq B^{i-1} \cdot g(M_B)$. We will only consider LBAs with the property that $B \leq g(M_B)$ and for any two tape sizes $B_1 \geq B_2$ we have $g(M_{B_1}) \geq g(M_{B_2})$. The LCL problem Π_M has time complexity $\Theta(n/B^{i-1}) = \Theta(g(M_B))$. The following theorem is proved in the full version of this paper.

▶ **Theorem 1.** *Problem Π_M has time complexity $\Theta(g(M_B))$.*

3.4.3 Instantiating the LCL construction

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

- By using a k-unary counter, for constant k, we obtain a growth of $\Theta(B^k)$.
- By using a binary counter, we obtain a growth of $\Theta(2^B)$.

▶ **Theorem 2.** *For any rational number $0 \leq \alpha \leq 1$, 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 with growth $\Theta(B^k)$ and using a $(j-k+1)$-dimensional grid graph, we obtain an LCL problem with complexity $\Theta(n/B^{j-k})$. We have that $n = \Theta(B^{j-k} \cdot g(M_B)) = \Theta(B^j)$, which implies $B = \Theta(n^{1/j})$. Thus the time complexity of our LCL problem is $\Theta(n/n^{(j-k)/j}) = \Theta(n^\alpha)$. ◀

▶ **Theorem 3.** *There exist LCL problems of complexities $\Theta(\frac{n}{\log^i n})$, for any positive integer i.*

Proof. Given an LBA with growth $\Theta(2^B)$ and using an $(i+1)$-dimensional grid graph, we obtain an LCL problem with complexity $\Theta(n/B^i)$. We have that $n = \Theta(B^i \cdot g(M_B)) = \Theta(B^i \cdot 2^B)$, which implies $B = \Theta(\log n)$. Thus the time complexity of our LCL problem is $\Theta(n/\log^i n)$. ◀

4 Complexity gap on trees

In this section we prove that, on trees, there are no LCLs having complexity T between $\omega(\sqrt{n})$ and $o(n)$. We show that, given an algorithm $\mathcal{A}$ that solves a problem in time T, it is possible to speed up its running time to $O(\sqrt{n})$, by first constructing a virtual tree S in which a ball of radius T corresponds to a ball of radius $O(\sqrt{n})$ of the original graph, and then find a valid output for the original graph, having outputs for the virtual graph S.

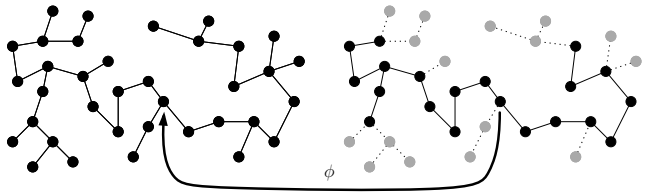

Figure 3 Example of a tree T and its skeleton T'; nodes removed from T in order to obtain T' are shown in gray. In this example, τ is 3.

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. We partition Ball_v, $\forall v \in V$, in d_v components (one for each neighbour of v), and let us denote these components with $C_i(v)$, where $1 \leq i \leq d_v$. 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 neighbor 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, $E' = \{e \in E(T) \mid e \text{ is not marked } \mathsf{Del}\}$, and $V' = \{u \in V \mid \exists\, w \in V \text{ s.t. } \{u, w\} \in E'\}$. Also, we want to keep track of the mapping from a node of T' to its original node in T; let ϕ 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 Figure 3 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. Informally, 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 ($c \leq l \leq 2c$) 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 [14].

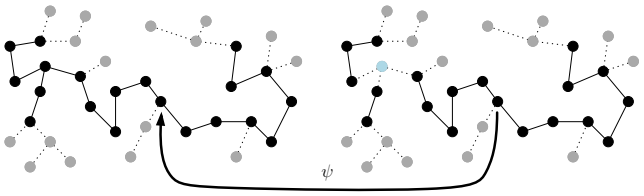

Figure 4 Example of the tree T'' obtained from T'; nodes with degree greater than 2 (in blue) are removed from T'.

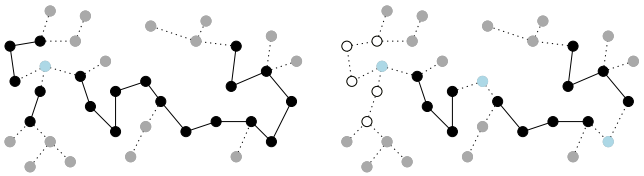

Figure 5 Blue nodes break the long paths $\mathcal{P}$ of T'' shown on the left into short paths $\mathcal{Q}$ shown in black on the right; short paths (in the example, paths with length less then 4) are ignored.

More formally, we start by splitting the tree in many paths of short length. Let v a node in V' and $d_v^{T'}$ its degree in T'. Let T'' be the forest obtained by removing from T' each node v having $d_v^{T'} > 2$. T'' is a collection $\mathcal{P}$ of disjoint paths. Let ψ be the mapping from nodes of T'' to their corresponding node in T'. See Figure 4 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 $c \leq l \leq 2c$, except for some paths of $\mathcal{P}$ that were already too short, or subpaths at the two ends of a longer path. Let $\mathcal{Q}$ be the subset of the resulting paths having length l satisfying $c \leq l \leq 2c$. See Figure 5 for an example.

In order to obtain the final tree, we use the following function, called Replace. Informally, given a graph G and a subgraph H 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 [8] in Section 3.3.

▶ **Definition 4** (Replace)**.** Let H be a subgraph of G. 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 $G' = \mathsf{Replace}(G, (H, F), (H', F'))$ is defined in the following way. Start 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. We will later define a function, Pump, that is used to obtain these longer paths. The function Pump is defined in an analogous way to the function Pump presented in [8] in Section 3.8. We now show which properties it satisfies.

▶ **Definition 5** (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 $\mathcal{T}_{\chi(v)}$, 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 $\mathsf{Pump}(Q^T, B)$ produces a new tree P^T having two endpoints, 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. Let

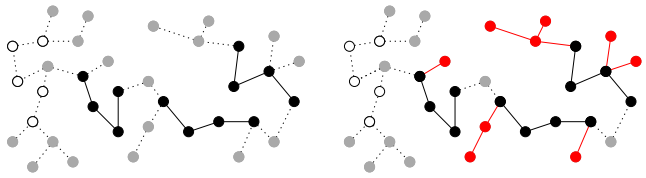

Figure 6 Example of $\mathcal{Q}^T$, obtained by merging the path nodes (in black) with previously removed trees connected to them (in red).

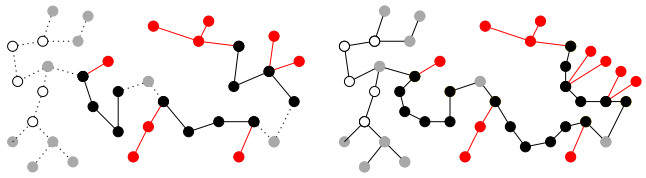

Figure 7 S (on the right) is obtained by pumping the black paths.

$G' = \mathsf{Replace}(G, (Q^T, (v_1, v_2)), (P^T, (v_1', v_2')))$. Pump satisfies that nodes $v_1', v_2' \in G'$ 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.

See Figure 6 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. Let $\mathcal{Q}^T$ be the set containing all Q^T. Replace each subgraph Q^T with $P^T = \mathsf{Pump}(Q^T, B)$. Note that a node v can not 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. 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 Figure 7 for an example.

Finally, we want to keep track of the *real* nodes of S. 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 $S_p = V(S) \setminus S_o$. 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. Let Virt be the function that maps T to S, that is, $S = \mathsf{Virt}(T, B, c)$. See Figure 8 for an example.

4.3 Properties of the virtual tree

The following lemma bounds the size of the graph S, compared to the size of T.

▶ **Lemma 6.** *The tree S has at most $N = c(B+1)n$ nodes, where $n = |V(T)|$, and $S = \mathsf{Virt}(T, B, c)$.*

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. ◀

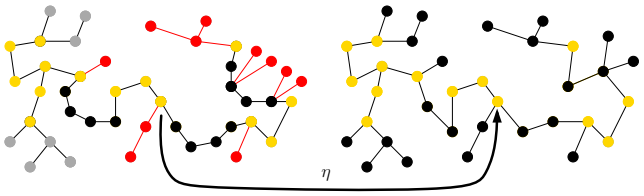

Figure 8 Nodes in yellow on the left are the ones in S_o, while the yellow ones on the right are nodes in T_o. Note that, for the sake of simplicity, we consider $2r = 1$.

The following lemma bounds the size of T' compared to the size of T''. 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 $\Theta(\sqrt{n})$ [8].

▶ **Lemma 7.** *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.

▶ **Lemma 8.** *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 7, 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 $\text{dist}(\phi^{-1}(u), \phi^{-1}(v)) \geq (c\sqrt{n}+1-(\frac{c\sqrt{n}+1}{c+1}+\frac{\sqrt{n}}{c}+1+\frac{\sqrt{n}}{c}(1+2(c-1))+2(c-1))) \cdot \frac{cB}{2c} - 1$, which is greater than $cB\sqrt{n}/3$ for c and n greater than a large enough constant. ◀

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 bruteforce. 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 can't see outside the subgraph of S constructed by v.

More precisely, all nodes do the following. First, they distributedly construct S, in $O(\sqrt{n})$ rounds. 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) and leaves out only nodes that correspond to pumped regions. Finally, nodes in $V(T) \setminus \Lambda$ find a valid output via bruteforce.

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 $V(T) \setminus \Lambda$ after having fixed outputs for nodes in Λ.

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

▶ **Lemma 9.** *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 6, 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 $\tau_{\mathsf{orig}}(N) \leq cB\sqrt{n}/3$, and since, by Lemma 8 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. ◀

4.5 Filling gaps by bruteforce

Using similar techniques presented in [8] we can 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. See the full version for the details.

References

1 Alkida Balliu, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Dennis Olivetti, and Jukka Suomela. New classes of distributed time complexity. In *Proc. 50th Annual Symposium on the Theory of Computing (STOC 2018)*. ACM, 2018 (to appear). arXiv:1711.01871.

2 Leonid Barenboim. Deterministic (Δ + 1)-coloring in sublinear (in Δ) time in static, dynamic, and faulty networks. *Journal of the ACM*, 63(5):47:1–47:22, 2016. doi:10.1145/2979675.

3 Leonid Barenboim, Michael Elkin, and Fabian Kuhn. Distributed ($\Delta+1$)-coloring in linear (in Δ) time. *SIAM Journal on Computing*, 43(1):72–95, 2014. doi:10.1137/12088848X.

4 Sebastian Brandt, Orr Fischer, Juho Hirvonen, Barbara Keller, Tuomo Lempiäinen, Joel Rybicki, Jukka Suomela, and Jara Uitto. A lower bound for the distributed Lovász local lemma. In *Proc. 48th Annual Symposium on the Theory of Computing (STOC 2016)*, pages 479–488. ACM, 2016. doi:10.1145/2897518.2897570.

5 Sebastian Brandt, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Patric R.J. Östergård, Christopher Purcell, Joel Rybicki, Jukka Suomela, and Przemysław Uznański. LCL problems on grids. In *Proc. 35th ACM Symposium on the Principles of Distributed Computing (PODC 2017)*, pages 101–110, 2017. doi:10.1145/3087801.3087833.

6 Yi-Jun Chang, Qizheng He, Wenzheng Li, Seth Pettie, and Jara Uitto. The complexity of distributed edge colouring with small palettes. In *Proc. 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2018)*. Society for Industrial and Applied Mathematics, 2018.

7 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the LOCAL model. In *Proc. 57th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2016)*, pages 615–624. IEEE, 2016. arXiv:1602.08166.

8 Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the LOCAL model. In *Proc. 58th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2017)*, 2017. arXiv:1704.06297.

9 Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986. doi:10.1016/S0019-9958(86)80023-7.

10 Pierre Fraigniaud, Marc Heinrich, and Adrian Kosowski. Local conflict coloring. In *Proc. 57th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2016)*, pages 625–634, 2016. doi:10.1109/FOCS.2016.73.

11 Mohsen Ghaffari and Hsin-Hao Su. Distributed degree splitting, edge coloring, and orientations. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*, pages 2505–2523. Society for Industrial and Applied Mathematics, 2017. doi:10.1137/1.9781611974782.166.

12 Juris Hartmanis and Richard Edwin Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965. doi:10.1090/S0002-9947-1965-0170805-7.

13 John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

14 Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. doi:10.1137/0221015.

15 Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

16 Alessandro Panconesi and Romeo Rizzi. Some simple distributed algorithms for sparse networks. *Distributed Computing*, 14(2):97–100, 2001. doi:10.1007/PL00008932.

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

18 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM Monographs on Discrete Mathematics and Applications. SIAM, Philadelphia, 2000.