Aalto University

Balliu, Alkida; Hirvonen, Juho; Korhonen, Janne H.; Lempiäinen, Tuomo; Olivetti, Dennis; Suomela, Jukka

New classes of distributed time complexity

New Classes of Distributed Time Complexity

Alkida Balliu
Aalto University, Finland
Gran Sasso Science Institute, Italy
Institut de Recherche en Informatique
Fondamentale, France
alkida.balliu@aalto.fi

Juho Hirvonen
University of Freiburg, Germany
IRIF, CNRS, and University Paris
Diderot, France
juho.hirvonen@cs.uni-freiburg.de

Janne H. Korhonen
Aalto University, Finland
janne.h.korhonen@aalto.fi

Tuomo Lempiäinen
Aalto University, Finland
tuomo.lempiainen@aalto.fi

Dennis Olivetti
Aalto University, Finland
Gran Sasso Science Institute, Italy
Institut de Recherche en Informatique
Fondamentale, France
dennis.olivetti@aalto.fi

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

ABSTRACT

A number of recent papers – e.g. Brandt et al. (STOC 2016), Chang et al. (FOCS 2016), Ghaffari & Su (SODA 2017), Brandt et al. (PODC 2017), and Chang & Pettie (FOCS 2017) – have advanced our understanding of one of the most fundamental questions in theory of distributed computing: what are the possible time complexity classes of LCL problems in the LOCAL model? In essence, we have a graph problem Π in which a solution can be *verified* by checking all radius-$O(1)$ neighbourhoods, and the question is what is the smallest T such that a solution can be *computed* so that each node chooses its own output based on its radius-T neighbourhood. Here T is the distributed time complexity of Π.

The time complexity classes for deterministic algorithms in bounded-degree graphs that are known to exist by prior work are $\Theta(1)$, $\Theta(\log^* n)$, $\Theta(\log n)$, $\Theta(n^{1/k})$, and $\Theta(n)$. It is also known that there are two gaps: one between $\omega(1)$ and $o(\log \log^* n)$, and another between $\omega(\log^* n)$ and $o(\log n)$. It has been conjectured that many more gaps exist, and that the overall time hierarchy is relatively simple – indeed, this is known to be the case in restricted graph families such as cycles and grids.

We show that the picture is much more diverse than previously expected. We present a general technique for engineering LCL problems with numerous different deterministic time complexities, including $\Theta(\log^\alpha n)$ for any $\alpha \geq 1$, $2^{\Theta(\log^\alpha n)}$ for any $\alpha \leq 1$, and $\Theta(n^\alpha)$ for any $\alpha < 1/2$ in the high end of the complexity spectrum, and $\Theta(\log^\alpha \log^* n)$ for any $\alpha \geq 1$, $2^{\Theta(\log^\alpha \log^* n)}$ for any $\alpha \leq 1$, and $\Theta((\log^* n)^\alpha)$ for any $\alpha \leq 1$ in the low end of the complexity spectrum; here α is a positive rational number.

CCS CONCEPTS

• **Theory of computation → Distributed computing models**; **Complexity classes**;

KEYWORDS

Distributed complexity theory, graph algorithms, locally checkable labellings, LOCAL model

1 INTRODUCTION

In this work, we show that the landscape of distributed time complexity is much more diverse than what was previously known. We present a general technique for constructing distributed graph problems with a wide range of different time complexities. In particular, our work answers many of the open questions of Chang and Pettie [7], and disproves one of their conjectures.

1.1 LOCAL Model

We explore here one of the standard models of distributed computing, the LOCAL model [21, 26]. In this model, we say that a graph problem (e.g., graph colouring) is solvable in time T if each node can output its own part of the solution (e.g., its own colour) based on its radius-T neighbourhood. We focus on deterministic algorithms – even though most of our results have direct randomised counterparts – and as usual, we assume that each node is labelled with an $O(\log n)$-bit unique identifier. We give the precise definitions in Section 2.

1.2 LCL Problems

The most important family of graph problems from the perspective of the LOCAL model is the class of LCL problems [22]. Informally, LCL problems are graph problems that can be solved in constant time with a *nondeterministic* algorithm in the LOCAL model, and the key research question is, what is the time complexity of solving LCL

A. Balliu, J. Hirvonen, J. H. Korhonen, T. Lempiäinen, D. Olivetti, J. Suomela

Table 1: State of the art: prior results on the existence of LCL problems in different complexity classes.

Complexity	Status	Reference
$O(1)$	exists	trivial
$\omega(1)$, $o(\log^* n)$	does not exist	[22]
$\Omega(\log^* n)$, $o(\log^* n)$	?	
$\Theta(\log^* n)$	exists	[8, 21]
$\omega(\log^* n)$, $o(\log n)$	does not exist	[6]
$\Theta(\log n)$	exists	[3, 6, 16]
$\omega(\log n)$, $n^{o(1)}$	?	
$\Theta(n^{1/k})$	exists	[7]
$\Theta(n)$	exists	trivial

problems with *deterministic* algorithms. Examples of LCL problems include the problem of finding a proper vertex colouring with k colours: if you guess a solution nondeterministically, you can easily verify it with a constant-time distributed algorithm by having each node check its local neighbourhood. As usual, we will focus on bounded-degree graphs. We give the precise definitions in Section 2.

1.3 State of the Art

Already in the 1990s, it was known that there are LCL problems with time complexities $O(1)$, $\Theta(\log^* n)$, and $\Theta(n)$ on n-node graphs [8, 21]. It is also known that these are the only possibilities in the case of cycles and paths [22]. For example, the problem of finding a 2-colouring of a path is inherently global, requiring time $\Theta(n)$, while the problem of finding a 3-colouring of a path can be solved in time $\Theta(\log^* n)$.

While some cases (e.g., oriented grids) are now completely understood [4], the case of general graphs is currently under active investigation. LCL problems with deterministic time complexities of $\Theta(\log n)$ [3, 6, 16] and $\Theta(n^{1/k})$ for all k [7] have been identified only very recently. It was shown by Chang et al. that there are no LCL problems with complexities between $\omega(\log^* n)$ and $o(\log n)$ [6]. Classical symmetry breaking problems like maximal matching, maximal independent set, $(\Delta+1)$-colouring, and $(2\Delta-1)$-edge colouring have complexity $\Theta(\log^* n)$ [1, 2, 13, 23]. Some classical problems are now also known to have intermediate complexities, even though tight bounds are still missing: Δ-colouring and $(2\Delta-2)$-edge colouring require $\Omega(\log n)$ rounds [3, 5], and can be solved in time $O(\text{polylog } n)$ [24]. Some gaps have been conjectured; for example, Chang and Pettie [7] conjecture that there are no problems with complexity between $\omega(n^{1/(k+1)})$ and $o(n^{1/k})$. See Table 1 for an overview of the state of the art.

The picture changes for randomised algorithms, especially in the region between $\omega(\log^* n)$ and $o(\log n)$. In the lower end of this region, it is known that there are no LCLs with randomised complexity between $\omega(\log^* n)$ and $o(\log \log n)$ [6], but e.g. sinkless orientations have a randomised complexity $\Theta(\log \log n)$ [3, 16], and it is known that no LCL problem belongs to this complexity class in the deterministic world. In the higher end of this region, it is known that all LCLs solvable in time $o(\log n)$ can be solved in time $T_{\text{LLL}}(n)$, the time it takes to solve a relaxed variant of the Lovász local lemma [7]. The current best algorithm gives $T_{\text{LLL}}(n) = 2^{O(\sqrt{\log \log n})}$ [11].

So far we have discussed the complexity of LCL problems in the strict classical sense, in graphs of maximum degree $\Delta = O(1)$. Many of these problems have been studied also in the case of a general Δ. The best deterministic algorithms for maximal independent set and $(\Delta+1)$-colouring run in time $2^{O(\sqrt{\log n})}$ [25]. Maximal matching can be solved in time $O(\log^2 \Delta \log n)$ [10] and $(2\Delta-1)$-edge colouring in time $O(\log^8 \Delta \log n)$ [12]. Corresponding randomised solutions are exponentially faster: $O(\log \Delta) + 2^{O(\sqrt{\log \log n})}$ rounds for maximal independent set [14], $O(\sqrt{\log \Delta}) + 2^{O(\sqrt{\log \log n})}$ for $(\Delta+1)$-colouring [18], $O(\log \Delta + \log^3 \log n)$ for maximal matching [10], and $2^{O(\sqrt{\log \log n})}$ for $(2\Delta-1)$-edge colouring [9, 12]. Some lower bounds are known in the general case: maximal independent set and maximal matching require time complexity of $\Omega(\min\{\log \Delta/\log \log \Delta, \sqrt{\log n/\log \log n}\})$ [20].

1.4 Contributions

Based on the known results related to LCL problems, it seemed reasonable to conjecture that there might be only three distinct non-empty time complexity classes below $n^{o(1)}$, namely $O(1)$, $\Theta(\log^* n)$, and $\Theta(\log n)$. There are very few candidates of LCL problems that might have any other time complexity, and in particular the gap between $\Omega(\log \log^* n)$ and $o(\log^* n)$ seemed to be merely an artefact of the current Ramsey-based proof techniques (see Chang and Pettie [7] for more detailed discussion on this region).

Our work changes the picture completely: we show how to construct *infinitely many* LCL problems for the regions in which the existence of any problems was an open question. We present a *general technique* that enables us to produce time complexities of the form $f(\log^* n)$ and $f(n)$ for a wide range of functions f, as long as f is sublinear and at least logarithmic. See Table 2 for some examples of time complexities that we can construct with our technique.

The table also highlights another surprise: the structure of "low" complexities below $O(\log^* n)$ and the structure of "high" complexities above $\omega(\log^* n)$ look now very similar.

1.5 Proof Ideas

On a high level, we start by defining a simple model of computation, called a *link machine* here. We emphasise that link machines are completely unrelated to distributed computing; they are simply a specific variant of the classical register machines. A link machine has $O(1)$ registers that can hold unbounded positive natural numbers, and a finite *program* (sequence of instructions). The machine supports the following instructions: resetting a register to 1, addition of two registers, comparing two registers for equality, and skipping operations based on the result of a comparison.

We say that a link machine M has *growth* $g \colon \mathbb{N} \to \mathbb{N}$ if the following holds: if we reset all registers to value 1, and then run the program of the machine M repeatedly for ℓ times, then the *maximum* of the register values is $g(\ell)$. For example, the following link machine has a growth $g(\ell) = \Theta(\ell^2)$:

```
x ← x + 1
y ← y + x
```

Table 2: Our contributions: examples of time complexity classes that are now known to contain an LCL problem. The integers r and s are positive constants.

"Low"			"High"		
$O(1)$		trivial	$\Theta(\log^* n)$		[8, 21]
$\big\vert$ *gap*		[7, 22]	$\big\vert$ *gap*		[6]
$\Theta(\log\log^* n)$		this work	$\Theta(\log n)$		[3, 6, 16]
$\Theta(\log^{r/s}\log^* n)$	$r/s \geq 1$	this work	$\Theta(\log^{r/s} n)$	$r/s \geq 1$	this work
$2^{\Theta(\log^{r/s}\log^* n)}$	$r/s \leq 1$	this work	$2^{\Theta(\log^{r/s} n)}$	$r/s \leq 1$	this work
			$\Theta(n^{1/s})$		[7]
$\Theta((\log^* n)^{r/s})$	$r/s \leq 1$	this work	$\Theta(n^{r/s})$	$r/s < 1/2$	this work
$\Theta(\log^* n)$		[8, 21]	$\Theta(n)$		trivial

Now assume that we have the following ingredients:

(1) A link machine M of growth g.
(2) An LCL problem Π for directed cycles, with a time complexity T.

We show how to construct a new LCL problem Π_M in which the "relevant" instances are graphs G with the following structure:

- There is a directed cycle C in which we need to solve the original problem Π.
- The cycle is augmented with an additional structure of multiple layers of "shortcuts", and the lengths of the shortcuts correspond to the values of the registers of machine M.

Therefore if we take ℓ steps away from cycle C, we will find shortcuts of length $g(\ell)$. In particular, if two nodes u and v are within distance $\ell \cdot g(\ell)$ from each other along cycle C, we can reach from u to v in $\Theta(\ell)$ steps along graph G.

In essence, we have compressed the distances and made problem Π_M easier to solve than Π, in a manner that is controlled precisely by function g. For example, if $g(\ell) = \Theta(\ell^2)$, then distance $\ell \cdot g(\ell) = \Theta(\ell^3)$ along C corresponds to distance $\Theta(\ell)$ in graph G. If Π had a time complexity of $T = \Theta(\log^* n)$, we obtained a problem Π_M with a time complexity of $T_M = \Theta((\log^* n)^{1/3})$.

Notice that these results could not be achieved by just adding shortcuts of length $g(n)$ directly onto every node of the cycle, since the lengths of the shortcuts would not be locally checkable, and moreover, it would not be true that a node can reach every other node within a certain distance.

1.6 Some Technical Details

Plenty of care is needed in order to make sure that

- Π_M is indeed a well-defined LCL problem: feasible solutions can be verified by checking the $O(1)$-radius neighbourhoods,
- Π_M is solvable in time $O(T_M)$ also in arbitrary bounded-degree graphs and not just in "relevant" instances that have the appropriate structure of a cycle plus shortcuts,
- there is no way to cheat and solve Π_M in time $o(T_M)$.

There is a fine balance between these goals: to make sure Π_M is solvable efficiently in adversarial instances, we want to modify the definition so that for unexpected inputs it is permitted to produce

the output "this is an invalid instance", but this easily opens loopholes for cheating – what if all nodes always claim that the input is invalid?

We address these issues with the help of ideas from *locally checkable proofs* and *proof labelling schemes* [17, 19], both for inputs and for outputs:

- Locally checkable inputs: Relevant instances carry a locally checkable proof. If the instance is not relevant, it can be detected locally.
- Locally checkable outputs: If the algorithm claims that the input is invalid, it also has to prove it. If the proof is wrong, it can be detected locally.

In essence, we define Π_M so that the algorithm has two possibilities in all local neighbourhoods: solve Π or prove that the input is invalid. This requirement can be now encoded as a bona fide LCL problem.

As a minor twist, we have slightly modified the above scheme so that we replace impossibility of cheating by hardness of cheating. Our LCL problem is designed so that an algorithm could, in principle, construct a convincing proof that claims that the input is invalid (at least for some valid inputs). However, to avoid detection, the algorithm would need to spend $\Omega(T_M)$ time to construct such a proof – in which case the algorithm could equally well solve the original problem directly.

1.7 Significance

The complexity classes and the gaps in the time hierarchy of LCL problems have recently played a key role in the field of distributed computing. The classes have served as a source of inspiration for algorithm design (e.g. the line of research related to the sinkless orientation problem [3, 6, 16] and the follow-up work [15] that places many other problems in the same complexity class), and the gaps have directly implied non-trivial algorithmic results (e.g. the problem of 4-colouring 2-dimensional grids [4]). The recently identified gaps [4, 6, 7, 11] have looked very promising; it has seemed that a complete characterisation of the LCL complexities might be within a reach of the state of the art techniques, and the resulting hierarchy might be sparse and natural.

In essence, our work shows that the free lunch is over. The deterministic LCL complexities in general bounded-degree graphs do not seem to provide any further gaps that we could exploit. Any of the currently known upper bounds might be tight. To discover new gaps, we will need to restrict the setting further, e.g. by studying restricted graph families such as grids and trees [4, 7], or by focusing on restricted families of LCL problems. Indeed, this is our main open question: what is the broadest family of LCL problems that contains the standard primitives (e.g., colourings and orientations) but for which there are large gaps in the distributed time hierarchy?

2 PRELIMINARIES

Let us first fix some terminology. We work with directed graphs $G = (V, E)$, which are always assumed to be simple, finite, and connected. We denote the number of nodes by $n = |V|$. The number of edges on a shortest path from node v to node u is denoted $\text{dist}(v, u)$. A *labelling* of a graph G is a mapping $l: V \to \Sigma$. Given a labelled graph (G, l), the *radius-T neighbourhood* of a node v consists of the subgraph $G_{v,T} = (V_{v,T}, E_{v,T})$, where $V_{v,T} = \{u \in V : \text{dist}(v, u) \leq T\}$ and $E_{v,T} = \{(u, w) \in E : \text{dist}(v, u) \leq T \text{ and } \text{dist}(v, w) \leq T\}$, as well as the restriction $l \restriction_{V_{v,T}}: V_{v,T} \to \Sigma$ of the labelling. The set of natural numbers is $\mathbb{N} = \{0, 1, \ldots\}$.

2.1 Model of Computation

Our setting takes place in the LOCAL model [21, 26] of distributed computing. We have a graph $G = (V, E)$, where each node $v \in V$ is a computational unit and all nodes run the same *deterministic* algorithm $\mathcal{A}$. We work with *bounded-degree graphs*; hence $\mathcal{A}$ can depend on an upper bound Δ for the maximum degree of G.

Initially, the nodes are not aware of the graph topology – they can learn information about it by communicating with their neighbours. To break symmetry, nodes have access to $O(\log n)$-bit unique identifiers, given as a labelling. We will also assume that the nodes are given as input the number n of nodes (for most of our results, e.g. a polynomial upper bound on n is sufficient). In addition, nodes can be given a task-specific local *input labelling*. We will often refer to directed edges, but for our purposes the directions are just additional information that is encoded in the input labelling. We emphasise that the directions of the edges do not affect communication; they are just additional information that the nodes can use.

The communication takes place in synchronous communication rounds. In each round, each node $v \in V$

(1) sends a message to each of its neighbours,
(2) receives a message from each of its neighbours,
(3) performs local computation based on the received messages.

Each node v is required to eventually halt and produce its own local output. We do not limit the amount of local computation in each round, nor the size of messages; the only resource of interest is the number of communication rounds until all the nodes have halted.

Note that in T rounds of communication, each node can gather all information in its radius-T neighbourhood, and hence a T-round algorithm is simply a mapping from radius-T neighbourhoods to local outputs.

2.2 Graph Problems

In the framework of the LOCAL model, the same graph $G = (V, E)$ serves both as the communication graph and the problem instance. In addition to the graph topology, the problem instance can contain local input labels. To solve the graph problem, each node is required to produce an output label so that all the labels together define a valid output.

More formally, let Σ and Γ be sets of input and output labels, respectively. A *graph problem* is a function $\Pi_{\Sigma,\Gamma}$ that maps each graph G and input labelling $i: V \to \Sigma$ to a set $\Pi_{\Sigma,\Gamma}(G, i)$ of valid solutions. Each solution is a function $o: V \to \Gamma$. We say that *algorithm $\mathcal{A}$ solves graph problem $\Pi_{\Sigma,\Gamma}$* if for each graph G, each input labelling $i: V \to \Sigma$ of G, and any setting of the unique identifiers, the mapping $o: V \to \Gamma$ defined by setting $o(v)$ to be the local output of node v for each $v \in V$, is in the set $\Pi_{\Sigma,\Gamma}(G, i)$. Note that the unique identifiers are given as a separate labelling; the set $\Pi_{\Sigma,\Gamma}(G, i)$ of valid solutions depends only on the task-specific input labelling i. When Σ and Γ are clear from the context, we denote a graph problem simply Π.

Let $T: \mathbb{N} \to \mathbb{N}$. Suppose that algorithm $\mathcal{A}$ solves problem Π, and for each input graph G, each input labelling i and any setting of the unique identifiers, each node needs as most $T(|V|)$ communication rounds to halt. Then we say that *algorithm $\mathcal{A}$ solves problem Π in time T*, or that the *time complexity* of $\mathcal{A}$ is T. The *time complexity of problem Π* is defined to be the slowest-growing function $T: \mathbb{N} \to \mathbb{N}$ such that there exists an algorithm $\mathcal{A}$ solving Π in time T.

In this work, we consider an important subclass of graph problems, namely *locally checkable labelling* (LCL) problems [22]. A graph problem $\Pi_{\Sigma,\Gamma}$ is an LCL problem if the following conditions hold:

(1) The label sets Σ and Γ are finite.
(2) There exists a LOCAL algorithm $\mathcal{A}$ having constant time complexity, such that given any labelling $l: V \to \Gamma$ as an additional input labelling, $\mathcal{A}$ can determine whether $l \in \Pi_{\Sigma,\Gamma}(G, i)$ holds: if $l \in \Pi_{\Sigma,\Gamma}(G, i)$, all nodes output "yes"; otherwise at least one node outputs "no".

That is, an LCL problem is one where the input and output labels are of constant size, and for which the validity of a candidate solution can be checked in constant time.

3 LINK MACHINES

A *link machine* M consists of a constant number k of *registers*, labelled with arbitrary strings, and a *program* P. The program is a sequence of instructions $i_1, i_2, \ldots, i_p$, where each instruction i_j is one of the following for some registers a, b, and c:

- Addition: $a \leftarrow b + c$.
- Reset: $a \leftarrow 1$.
- Conditional execution: If $a = b$ (or if $a \neq b$), execute the next s instructions, otherwise skip them.

The registers can store unbounded natural numbers. For convenience, we will generally identify the link machine with its program.

An *execution* of the link machine M is a single run through the program, modifying the values of the registers according to the instructions in the obvious way. Generally, we consider computing with link machines in a setting where

- all registers start from value 1, and
- we are interested in the maximum value over all registers after ℓ executions of M.

Specifically, for a register r, we denote by $r(\ell)$ the value of register r after ℓ full executions of the link machine program, starting from all registers set to 1. We say that a link machine M with registers $r_1, r_2, \ldots, r_k$ has *growth* $g\colon \mathbb{N} \to \mathbb{N}$ if, starting from all registers set to 1, we have that $\max\{r_i(\ell)\colon i = 1, 2, \ldots, k\} = g(\ell)$ for all $\ell \in \mathbb{N}$. While g does not need to be a bijection, we use the notation $g^{-1}\colon \mathbb{N} \to \mathbb{N}$ to denote the function defined by setting $g^{-1}(\ell) = \min\{m \in \mathbb{N} \colon g(m) \geq \ell\}$ for all $\ell \in \mathbb{N}$.

3.1 Working with Link Machine Programs

Composition. Consider two link machines M_1 and M_2 with corresponding programs P_1 and P_2. By relabelling if necessary, we can assume that the programs do not share any registers. Moreover, assume P_1 has a register y we call the *output register* for P_1 and P_2 has a register x we call the *input register* for P_2. We define the *composition* $P_2 \circ P_1$ as the program

```
P_1
x ← y
P_2
```

Note that the growth of the program $P_2 \circ P_1$ at step ℓ is the maximum between the growth of each program at step ℓ, and can be affected by the input given by P_1 to P_2. The basic idea is to use this construct so that P_1 produces an output register $y(\ell)$ dependent on ℓ, which is then used by P_2 to produce a composed growth function $g(y(\ell))$.

We define the composition $P_i \circ \cdots \circ P_2 \circ P_1$ of multiple link machine programs with specified input and output registers similarly.

Note that the growth of a link machine is at most $2^{O(\ell)}$. As we will see later this constraint is necessary, since otherwise we would contradict known results regarding gaps on LCL complexities.

3.2 Building Blocks

We now define our basic *building blocks*, that is, small programs that can be composed to obtain more complicated functions. These building blocks are summarised in Table 3. In all our cases, we will assume that the value of the input register x is growing and bounded above by ℓ; otherwise the semantics of a building block is undefined.

Link machine programming conventions. We use the following shorthands when writing link machine programs:

- We write conditional executions as IF-THEN constructs, with the conditional execution skipping all enclosed instructions if the test fails. We also use IF-ELSE constructs, as these can be implemented in an obvious way.
- We write sums with multiple summands, constant multiplications, and constant additions as single instructions, as these can be easily simulated by multiple instructions and a constant number of extra registers.

Table 3: Our basic building blocks. The integer k is a constant. Programs with no input generate output values that only depend on the number of executions ℓ. Programs with input assume that the value of the input register x is growing and bounded above by ℓ.

Program P	Input	Output	Growth
COUNT	–	$y = \ell$	ℓ
ROOT$'_k$	–	$y = \Theta(\ell^{1/k})$	$\Theta(\ell^{1/k})$
ROOT$_k$	x	$y = \Theta(x^{1/k})$	$\Theta(x)$
POW$_k$	x	$y = \Theta(x^k)$	$\Theta(x^k)$
EXP	x	$y = 2^{\Theta(x)}$	$2^{\Theta(x)}$
LOG	x	$y = \Theta(\log x)$	$\Theta(x)$

Counting. Our first program COUNT simply produces a linear output $y = \ell$:

```
y ← y + 1
```

Clearly, program COUNT has growth ℓ.

Polynomials. Next, we define a sequence of programs for computing $y = \Theta(x^k)$. For any fixed $k \geq 1$, we define the program POW$_k$ as follows:

```
if  x ≠ x_1  then
    x_k ← Σ_{i=0}^{k} (k choose i) x_i
    x_{k-1} ← Σ_{i=0}^{k-1} (k-1 choose i) x_i
    · · ·
    x_1 ← x_1 + 1
end if
y ← x_k
```

We now have that $x_1 = \Theta(x)$, and by the binomial theorem, $x_i = (x_1)^i$ for all $i = 1, 2, \ldots, k$. Moreover, POW$_k$ has growth $\Theta(x^k)$.

Roots. We define two versions of a program computing a kth root. The first one does not take an input and has the advantage that it has sublinear growth of $\Theta(\ell^{1/k})$. Specifically, we define ROOT$'_k$ as follows:

```
if  y_1 ≠ y_2  then
    y_1 ← y_1 + 1
else if  y_2 ≠ y_3  then
    y_2 ← y_2 + 1
    y_1 ← 1
else if  . . .  then
    . . .
else if  y_{k-1} ≠ y_k  then
    y_{k-1} ← y_{k-1} + 1
    y_1 ← 1, y_2 ← 1, . . . , y_{k-2} ← 1
else
    y_k ← y_k + 1
```

```
        y₁ ← 1, y₂ ← 1, . . . , y_{k−1} ← 1
      end if
      y ← y_k
```

Observe that started from all registers set to 1, we always have $y_1 \leq y_2 \leq \ldots \leq y_k$. Moreover, for register y_k to increase from s to $s+1$, the values of the registers y_i will visit all configurations where $y_1 \leq y_2 \leq \ldots y_{k-1} \leq s$, and there are $\binom{s+k-2}{k-1}$ such configurations. This implies that the growth of register y_k is $\Theta(\ell^{1/k})$.

The second version of the kth root program takes an input register x, and computes an output $y = \Theta(x^{1/k})$. We define this program ROOT_k as follows:

```
  if  x ≠ x′  then
      x′ ← x′ + 1
      ROOT′_k
  end if
```

Clearly, we have that $x' = \Theta(x)$, and by the properties of ROOT'_k the output register is $y = \Theta(x^{1/k})$. The growth of ROOT_k is $g(\ell) = x$.

Exponentials. The program EXP computes an exponential function in the input register x:

```
  if  x ≠ x′  then
      y ← y + y
      x′ ← x′ + 1
  end if
```

We have that $x' = \Theta(x)$, and $y = 2^{x'} = 2^{\Theta(x)}$. Moreover, the growth of EXP is $2^{\Theta(x)}$.

Logarithms. The program LOG computes a logarithm of the input register x:

```
  if  x ≠ x′  then
      if  x′ = z  then
          z ← z + z
          y ← y + 1
      end if
      x′ ← x′ + 1
  end if
```

Clearly, we have that $x' = \Theta(x)$, and $z = \Theta(x')$. Starting from the valid starting configuration, the register z only takes values that are powers of two, and $y = \log_2 z$. Thus, we have $y = \Theta(\log x)$. By construction, the growth of LOG is $g(\ell) = z = \Theta(x)$.

3.3 Composed Functions

By composing our building block functions, we can now construct more complicated functions, which will then be used to obtain LCL problems of various complexities. The constructions we use are listed in Table 4; the values of output registers and the functions

Table 4: Composed programs. The integers p and q are constants.

Program P		Growth
$\text{POW}_p \circ \text{ROOT}'_q$		$\Theta(\ell^{p/q})$
$\text{EXP} \circ \text{POW}_q \circ \text{ROOT}'_p$	$(p \geq q)$	$2^{\Theta(\ell^{q/p})}$
$\text{EXP} \circ \text{POW}_q \circ \text{ROOT}_p \circ \text{LOG} \circ \text{COUNT}$	$(p \geq q)$	$2^{\Theta(\log^{q/p} \ell)}$

computed by these programs follow directly from the results in Section 3.2.

Notice that for all the considered programs there is a register that is always as big as all the other registers. Thus, we can refer to it as the register of maximum value.

Remark 1. While exploring the precise power of link machines is left as an open question, we point out that, in this paper, we do not list every possible complexity that one can achieve with link machines. Indeed, there are many more time complexities that can be realised; for example, one could define a building block that performs a multiplication, or add support for negative numbers and subtractions.

4 LINK MACHINE ENCODING GRAPHS

In this section, we show how to encode link machine executions as locally checkable graphs. Fix a link machine M with k registers and a program of length p that has non-decreasing growth g in $\omega(1)$ and $2^{O(n)}$, and let h be an integer. The basic idea is that we encode the link machine computation of the value $g(h)$ as follows:

- We start from an $h \times n$ grid graph, where $n = 3g(h)$, that wraps around in the horizontal direction, as shown in Figure 1 (here 3 is the smallest constant that avoids parallel edges or self-loops). This allows us to 'count' in two dimensions; one is used for time, and the other for the values of the registers of M. The grid is consistently oriented so that we can locally distinguish between the two dimensions, and all grid edges are labelled with either 'up', 'down', 'left' or 'right' to make this locally checkable.
- We add horizontal edges to the grid graph to encode the values of the registers. Specifically, at level ℓ of the graph, the horizontal edges encode the values the registers take during the ℓth execution of the link machine program, with edge labels specifying which register values the edges are encoding (see Figure 2).

The labels should be thought as LCL input labels; as we will see later, they will allow us to recognise valid link machine encoding graphs in the sense of locally checkable proofs. We will make this construction more formal below.

4.1 Formal Definition

Let M be a link machine with growth g as above. We formally define the *link machine encoding graphs* for M as graphs obtained from the construction we describe below.

Grid structure. The construction starts with a 2-dimensional $h \times n$ grid graph, where $n = 3g(h)$. Let (x, ℓ) denote the node on the
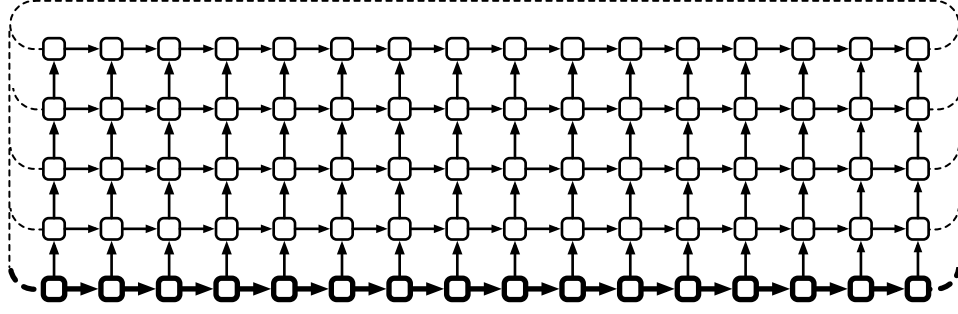
Figure 1: The base grid. The bottom cycle (first level) is highlighted.

ℓth row and the xth column, where $x \in \{1, 2, \ldots, n\}$ and $\ell \in \{1, 2, \ldots, h\}$. The grid wraps around along the horizontal axis, that is, we also add the edges $\big((n, \ell), (1, \ell)\big)$ for all ℓ.

We add horizontal *link edges* to the graph according to the state of the machine M. That is, we say that for a node (x, ℓ), a *link edge of length s* is an edge $\big((x, \ell), (x + s \bmod n, \ell)\big)$. Let $r(\ell, i)$ denote the value of the register r after executing $\ell - 1$ times the full program of M, and then executing the first i instructions of M. For each $\ell = 1, 2, \ldots, h$, register r, and $i = 0, 1, 2, \ldots, p$, we add a link edge of length $r(\ell, i)$ to all nodes on level ℓ if it does not already exist.

Local labels. In addition to the graph structure, we add *constant-size* labels to the graph as follows. First, each node (x, ℓ) has a set of labels for each incident edge, added according to the following rules if the corresponding edge is present (note that a single edge may have multiple labels):

- The grid edge to $(x, \ell + 1)$ is labelled with U.
- The grid edge from $(x, \ell - 1)$ is labelled with D.
- The grid edge from $(x - 1 \bmod n, \ell)$ is labelled with L.
- The grid edge to $(x + 1 \bmod n, \ell)$ is labelled with R.
- For each register r and $i = 0, 1, 2, \ldots, p$, the link edge of length $r(\ell, i)$ is labelled with (r, i).

Consider the set of labels that each node (x, ℓ) associates to each of its incident edges. When we later define graph problems, we assume these labels to be implicitly encoded in the node label given to (x, ℓ).

*Input.* Also, each node v is provided with an input $i(v) \in \{0, 1\}$.

4.2 Local Checkability

We show that the labels described in Section 4.1 constitute a *locally checkable proof* for the graph being a link machine encoding graph. That is, there is a LOCAL algorithm where all nodes accept a labelled graph if and only if it is a link machine encoding graph.

Local constraints. We first specify a set of local constraints that are checked by the nodes locally. All the constraints depend on the radius-4 neighbourhood of the nodes, so this can be implemented in the LOCAL model in 4 rounds. In the following, for labels $L_1, L_2, \ldots, L_k$, let $v(L_1, L_2, \ldots, L_k)$ denote the node reached by following the edges with the specified labels. The full constraints are now as follows:

(1) Each node checks that the basic properties of the labelling are correct:
   - All possible edge labels are present exactly once, except possibly one of D and U.
   - The direction labels U, D, L, and R are on different edges if present.
(2) Grid constraints ensure the validity of the grid structure:
   - Each node checks that each of the edges labelled with U, D, L or R has the opposite label in the other end.
   - If there is an edge labelled D, check that $v(\mathsf{D}, \mathsf{R}, \mathsf{U}) = v(\mathsf{R})$.
   - If there is not an edge labelled D, check that also nodes $v(\mathsf{L})$ and $v(\mathsf{R})$ do not have edges labelled D.
   - If there is not an edge labelled U, check that also nodes $v(\mathsf{L})$ and $v(\mathsf{R})$ do not have edges labelled U.
(3) Nodes check that the values of the registers are correctly initialised on the link edges:
   - Nodes that do not have an edge labelled with D check that the register values are initialised to 1, that is, the labels R and $(r, 0)$ are on the same edge for all registers r.
   - Nodes that have an edge labelled with D check that the registers are copied correctly, that is, for all registers r, $v((r, 0)) = v(\mathsf{D}, (r, p), \mathsf{U})$.
(4) Nodes check that the program execution is encoded correctly as follows. Each instruction is processed in order, from 1 to p. The ith is checked as follows, depending on the type of the instruction:
   - If the instruction is $a \leftarrow 1$:
   (a) Register a is correctly set to 1: the labels R and (a, i) are on the same edge.
   (b) Any of the other registers did not change, that is, labels $(r, i - 1)$ and (r, i) are on the same edge for all registers r except a.
   - If the instruction is $a \leftarrow b + c$:
   (a) Register a is set correctly:

$$v\big((a, i)\big) = v\big((b, i - 1), (c, i - 1)\big).$$

   (b) Any of the other registers did not change, that is, labels $(r, i - 1)$ and (r, i) are on the same edge for all registers r except a.
   - If the instruction is an IF statement comparing registers a and b, check if the labels $(a, i - 1)$ and $(b, i - 1)$ are on the
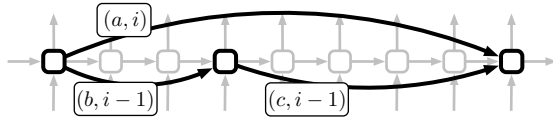
Figure 2: Checking the correct encoding of the execution of the instruction $a \leftarrow b + c$.

same edge, and if this does not match the condition of the IF statement, check that the following s instructions are not executed:

(a) Any registers do not change for s steps, that is, for all registers r, we have that labels $(r, i-1), (r, i), (r, i+1), \ldots, (r, i+s)$ are on the same edge.

(b) Skip the checks for the next s instructions.

(5) If no edges are labelled U, check that the link edges corresponding to the register with the maximum value form 3-cycles.

Correctness. It is clear that link machine encoding graphs satisfy the constraints (1)–(5) specified above. Conversely, we want to show any graph satisfying these constraints is a link machine encoding graph, but it turns out this is not exactly the case.

It might happen that the register values exceed the width w of the grid, the edges "wrap around", and the correspondence between the edge lengths and register values gets lost. However, for this to happen one has to have a row ℓ with $g(\ell) \geq w$.

In order to characterise the graph family captured by the local constraints, we define that a graph G is an *extended* link machine encoding graph if

- G is an $h \times w$ grid for some h and w that wraps around horizontally but not vertically,
- G satisfies the *local* constraints of link machine encoding graphs, and
- there is an $\ell \leq h$ with $g(\ell) \geq w/3$ such that up to row $\ell - 1$ the edge lengths are in one-to-one correspondence with the register values of the first $\ell - 1$ executions of link machine M.

Note that a link machine encoding graph is trivially an extended link machine encoding graph, as we can simply choose $\ell = h$ and $w = n$ and hence $g(\ell) = w/3$. The intuition is that extended link machine encoding graphs have a good bottom part with dimensions $\ell \times \Theta(g(\ell))$, and on top of that there might be any number of additional rows of some arbitrary garbage.

Lemma 2. *Let G be a graph that satisfies the constraints (1)–(5) and has at least one node that does not have an edge labelled with* D. *Then G is an extended link machine encoding graph.*

PROOF. By constraints (1) and (2), we have that the graph G is a grid graph that wraps around horizontally. By the assumption that there is a node without edge labelled by D and by constraint (2), the grid cannot wrap around vertically. Hence G is an $h \times w$ grid that wraps around horizontally, for some values h and w, and by assumption it satisfies the local constraints of link machine encoding graphs.

Constraints (3) and (4) ensure that the link edges and the corresponding labels are according to the link machine encoding graph specification, as long as $g(\ell) \leq w$.

Constraint (5) ensures that we cannot have $g(\ell) < w/3$ for all ℓ, as in the top row we must have three edges that form a cycle that wraps around the entire grid of width w at least once. Hence at some point we must reach $g(\ell) \geq w/3$, and this is sufficient for G to satisfy the definition of an extended link machine encoding graph.   □

5   LCL CONSTRUCTIONS

Let M be a link machine with non-decreasing growth g in $\omega(1)$ and $2^{O(n)}$, and let Π be an LCL problem on directed cycles with complexity $T(n)$ – for concreteness, Π will either be 3-colouring (complexity $\Theta(\log^* n)$) or a variant of 2-colouring (complexity $\Theta(n)$). To simplify the construction, we will assume that Π is solvable on directed cycles with *one-sided* algorithms, i.e., with algorithms in which each node only looks at its $T(n)$ successors. We now construct an LCL problem Π_M with complexity related to g, as outlined in the introduction:

- If a node sees a graph that locally looks like a link machine encoding graph, and the node is on the bottom row of the grid, it will need to solve problem Π on the directed cycle formed by the bottom row of the grid. As will be shown later, in $\Theta(\ell)$ steps, a node on the bottom row of the grid sees all nodes within distance $f(\ell) = \ell g(\ell)$ on the bottom cycle, so this is solvable in $\Theta(f^{-1}(T(n)))$ rounds.

- If a node sees something that does not look like a link machine encoding graph, it is allowed to report an error; the node must also provide an *error pointer* towards an error it sees in the graph. A key technical point here is to ensure that it is not too easy to claim that there is an error somewhere far, even if in reality we have a valid link machine encoding graph. We address this by ensuring that error pointer chains can only go right and then up, they cannot disappear without meeting an error, the part that is pointing right must be properly 2-coloured, and the part that is pointing up copies the input $i(v)$ given to the node v that is witnessing the error. If some nodes claim that the error is somewhere far up, we will eventually reach the highest layer of the graph and catch the cheaters. Also, nodes cannot blindly point up, because they need to mark themselves with the input of the witness. If all bottom-level nodes claim that the error is somewhere right, we do not necessarily catch cheaters, but the nodes did not gain anything as they had to produce a proper 2-colouring for the long chain of error pointers.

There are some subtleties in both of these points, which we address in detail below.

5.1   The LCL Problem Π_M

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 link machine encoding graph labelling for M as described in Section 4. The possible output labels are the following:

(1) output labels of the LCL problem Π,
(2) an *error label* E,

(3) an *error pointer*, pointing either right (R) or up (U), with a counter mod 2 and a label c_e in $\{0, 1\}$,

(4) an empty output ϵ.

The correctness of the output labelling is defined as follows.

(1) If the input labelling does not locally satisfy the constraints of link machine encoding graphs for M (see Section 4.2) either at the node itself or at one of its neighbours, then the only valid output is E. Otherwise, the node must produce one of the other labels.

(2) If the output of a node v is one of the labels of Π, then the following must hold:
   - Node v does not have an incident edge with label D in the input labelling.
   - If any adjacent nodes have output from the label set of Π, then the local constraints of Π must be satisfied.

(3) If the output of a node v is empty, then it must have an incident edge with label D in the input labelling.

(4) If the output of a node v is an error pointer, then the following must hold:
   - Node v has only one outgoing error pointer.
   - The error pointer is pointing either R or U if the node does not have an edge labelled D in the input, and U if the node does have an edge labelled D in the input.
   - The node at the other end of the pointer either outputs an error label E, or an error pointer.
   - The mod-2 counters of nodes outputting error pointer R form a 2-colouring in the induced subgraph of those nodes.
   - The nodes v outputting error pointer U have the same label $c_e(v)$ as the next node in the chain. If v is the last node in the chain, $c_e(v) = i(w)$ holds, where w is the witness outputting E.

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

5.2 Time Complexity

We now prove the following bounds for the time complexity of problem Π_M; recall that $f(k) = kg(k)$, where g is the growth of link machine M. In the following, n denotes the number of nodes in the input graph, and $\hat{n}$ is the smallest number satisfying $n \le \hat{n} \cdot f^{-1}(T(\hat{n}))$. The intuition here is that the "worst-case instances" of size n will be grids of width approximately $\hat{n}$ and height approximately $f^{-1}(T(\hat{n}))$.

Theorem 3. *Problem Π_M can be solved in $O\big(f^{-1}\big(T(\hat{n})\big)\big)$ rounds.*

Theorem 4. *Problem Π_M cannot be solved in $o\big(f^{-1}\big(T(\hat{n})\big)\big)$ rounds.*

5.2.1 Upper Bound – Proof of Theorem 3. We start by observing that the link machine encoding graphs essentially provide a 'speed-up' in terms of how quickly the nodes on the bottom cycle can see other nodes on the bottom cycle. Recall that $f(k) = kg(k)$, where g is the growth of link machine M; also recall that $g(k) = 2^{O(k)}$.

Lemma 5. *Let G be a link machine encoding graph for M, and let u and v be nodes on the bottom cycle. If u can be reached from v in ℓ steps following edges labelled with R, then u can be reached from v in $O\big(f^{-1}(\ell)\big)$ steps following edges labelled with U, D, R or register labels.*

Proof. Starting from a node v on the cycle, in $O(k)$ steps it is possible to see a node on the cycle that is $f(k) = kg(k)$ steps away: take k steps up, k steps right along shortcuts, and k steps down. We will use a similar procedure to go to a node at any distance ℓ.

Let k be the smallest value such that $f(k) \ge \ell$. By assumption, $f(k - 1) = (k - 1)g(k - 1) < \ell$. Since $g(k) = 2^{O(k)}$, we have $g(k)/g(k - 1) \le k - 1$ for large enough graphs, and hence $g(k) \le (k - 1)g(k - 1) = f(k - 1) < \ell$.

We find a path P from v to u by a greedy procedure. First go up for k steps. Recall that at height k there are shortcuts of length $g(k)$. Go right along shortcuts until the distance to the column of u is less than $g(k)$ (taking one more shortcut would bring us to a node that is on the right of u). This takes at most k steps. Next step down and do a greedy descent to get to the column of u. At each level h, if the remaining distance to the column of u is at least $g(h)$, take steps along the longest shortcut until the distance is less than $g(h)$. Since $g(k) = 2^{O(k)}$, the length of the shortcuts at level h is at most a constant number of times the length at level $h - 1$, hence this number of steps is bounded by $O(1)$. Finally, step down. We either reach the column of u or the bottom cycle, and in this case the distance to the column of u is less than $g(1)$.

Since $f(k - 1) < \ell$, then $k < f^{-1}(\ell) + 1$. We take a total of at most k steps up and down, at most k steps right at level k, and $O(1)$ steps for each of the k levels, for a total of $O(f^{-1}(\ell))$ steps. □

Lemma 6. *If a node not having an edge labelled D sees no errors within distance $r = Cf^{-1}(T(\hat{n}))$ for a sufficiently large constant C, then it can produce a valid output for the problem Π.*

Proof. First consider a global problem, i.e., $T(n) = \Theta(n)$. If we explore the grid up and right for $r = Cf^{-1}(\hat{T}(n))$ steps, and we do not encounter any errors, and the grid does not wrap around, then we would discover a grid fragment of dimensions at least $Df^{-1}(\hat{n}) \times D\hat{n}$ for a $D = \Omega(C)$. Such a grid fragment would contain $D^2 n$ nodes, and for a sufficiently large D this would contradict the assumption that the input has n nodes. Hence we must encounter errors (which by assumption is not the case), or the grid has to wrap around cleanly without any errors, in which case we also see the entire bottom row and we can solve Π there by brute force.

Second, consider the case where $T(n) = \Theta(\log^* n)$. By a similar reasoning, the node can gather a grid fragment of dimensions $Df^{-1}(\log^* \hat{n}) \times D\log^* \hat{n}$. In particular, it can see a fragment of length $D\log^* \hat{n}$ of the bottom row. Furthermore, we have $\log^* n = O(\log^* \hat{n})$: to see this, note that g is non-decreasing, $f(k) \ge k$, and hence $n = o(\hat{n}^2)$. Therefore in r rounds, for a sufficiently large C, we can gather a fragment of the bottom row that spans up to distance at least $T(n)$, and this is enough to solve Π. □

As a consequence, we obtain an upper bound for the complexity of Π_M.

Theorem 3. *Problem Π_M can be solved in $O\big(f^{-1}\big(T(\hat{n})\big)\big)$ rounds.*

Proof. The idea of the algorithm that solves the described LCL problem is the following. First, each node gathers its constant-radius neighbourhood, and sees if there is a local error:
   - If a node witnesses a local error, it marks itself as 'witness'
   - If a node is either a witness itself, or it is adjacent to a witness, it marks itself as a *near-witness*, and outputs E.

Now let $r = c\,f^{-1}(T(\hat{n}))$, for a large enough constant c. Each node v in the bottom cycle – not having an edge labelled D – attempts to gather full information about an $r \times f(r)$ rectangle to the up and right from node v, that is, a rectangle composed by the bottom-most r nodes of the first $f(r)$ columns to the right of v. By Lemma 5, in $O(r)$ rounds we can either successfully gather the entire rectangle if it is error-free, or we can discover the nearest column that contains a near-witness:

(1) If the entire rectangle is error-free, we can solve Π on the bottom row by Lemma 6.

(2) Otherwise, we find the nearest column containing a near-witness $\bar{w}$. In such a case, node v will output its modulo-2 distance from that column, the input $i(\bar{w})$ of the witness, and produce a path of error pointers that spans a sequence of edges labelled R followed by a sequence of edges labelled U, reaching $\bar{w}$. Notice that this path is unique and always exists, since all columns before the nearest one containing a witness must be fault-free (up to height r), and if a witness is in the same column of v, the lowest one can be always reached by a fault-free path spanning only edges labelled U.

Finally, nodes that are not on the bottom cycle and do not see bottom nodes wanting to produce error pointer paths produce empty outputs.

Clearly, this produces a valid solution to Π_M on extended link machine encoding graphs, since they satisfy Lemma 6. Also, if there are no witnesses and every node has an edge labelled D, all nodes produce empty outputs, that is valid.

Now, consider a graph that is not an extended link machine encoding graph. A node will explore the graph for $\Theta(r)$ rounds. If the node satisfies the requirements of Lemma 6, then it produces a valid solution for the problem Π. Otherwise the node sees a witness. If a node v decides to produce an error pointer towards a near-witness $\bar{w}$, then all the nodes on the error path will produce an error pointer towards $\bar{w}$. This follows from the observation that, on valid fragments, nodes on the same row reach the same height while visiting the graph, due to the rectangular visit. Thus, if v outputs a pointer towards $\bar{w}$, then all the intermediate nodes will output a pointer, and these pointers will correctly produce a path from v to $\bar{w}$ with the right modulo-2 distance and c_w labelling. □

5.2.2 *Lower Bound – Proof of Theorem 4.* Next, we prove that the upper bound in Theorem 3 is tight. The worst-case instances are *truncated* link machine encoding graphs, defined as follows: take a valid link machine encoding graph and remove rows from the top, until it is satisfied that $n \leq \hat{n} \cdot f^{-1}(T(\hat{n}))$, where $\hat{n}$ is the length of the bottom cycle. The basic idea is to show that on truncated link machine encoding graphs, any algorithm has two choices, both of them equally bad:

• We can solve problem Π on the bottom cycle, but this requires time $\Omega(f^{-1}(T(\hat{n})))$.

• We can report an error, but this also requires $\Omega(f^{-1}(T(\hat{n})))$ rounds.

Note that truncated link machine encoding graphs have errors, and hence it is fine for a node to report an error. However, all witnesses are on the top row (or next to it), and constructing a correctly labelled error pointer chain from the bottom row to the top row takes time linear in the height of the construction. We will formalise this intuition in what follows.

Lemma 7. *Let G be a truncated link machine encoding graph, v be a node on the bottom row, and let h be any function satisfying $h(\ell) = o(f^{-1}(\ell))$. Let X be the set of all nodes that v can see in $h(\ell)$ steps. Then X is contained in the subgraph induced by the columns within distance $o(\ell)$ of v.*

PROOF. By the construction of link machine encoding graphs, the maximum distance in columns we can reach in ℓ steps is bounded by $f(\ell)$. Since $f(\ell) = \omega(\ell)$, we have that $f(\ell/C) \leq f(\ell)/C$ for any positive integer C. Thus, for any C, there is ℓ_0 such that $h(\ell) < f^{-1}(\ell)/C$ for all $\ell \geq \ell_0$, and thus

$$f(h(\ell)) < f(f^{-1}(\ell)/C) \leq \ell/C$$

for all $\ell \geq \ell_0$, which implies the claim. □

Theorem 4. *Problem Π_M cannot be solved in $o(f^{-1}(T(\hat{n})))$ rounds.*

PROOF. The nodes on the bottom row have the following possible outputs:

(1) At least one node v produces an error pointer U. Then we must have a chain of U pointers all the way to the near-witness w near the top row, and the chain has to be labelled with the input of w. The distance from v to w is $\Theta(f^{-1}(T(\hat{n})))$, and the claim follows.

(2) None of the nodes on the bottom row produce an error pointer U, but at least one of them produces an error pointer R. But then all nodes on the bottom row must output R, and the bottom cycle has to be properly 2-coloured.

(3) None of the nodes produce any error pointers. Then all nodes on the bottom row must solve problem Π.

As 2-colouring the bottom row is at least as hard as solving problem Π on the bottom row, it is sufficient to argue that the third case requires $\Omega(f^{-1}(T(\hat{n})))$ rounds. The proof is by simulation. We assume a faster algorithm for Π_M and use it to speed up the corresponding problem Π on cycles.

Let A be an algorithm for Π_M with running time $o(f^{-1}(T(\hat{n})))$. The algorithm has to solve the problem Π on the bottom cycle. Now, given a cycle C of length $\hat{n}$ as input, we create a virtual link machine encoding graph on top of the cycle as follows: each node creates the nodes in its column, their identifiers defined to be the identifier of the bottom node padded with the node's height, encoded in $\log h$ bits.

To simulate an algorithm with running time $t = o(f^{-1}(T(\hat{n})))$ in this virtual graph, each node needs to learn the identifiers of all nodes in its radius-t neighbourhood in the virtual graph. By Lemma 7, the columns of those nodes are contained within distance $o(T(\hat{n}))$ in the virtual graph. Thus, we can recover the identifiers of the nodes by scanning the cycle C up to distance $o(T(\hat{n}))$. Now each node v can apply A and find a solution for Π on the cycle C in time $o(T(\hat{n}))$, by outputting the output of the node at the bottom of the virtual column created by node v. This yields an algorithm with running time $o(T(\hat{n}))$ on cycles of length $\hat{n}$, a contradiction. □

Remark 8. Note that we did not use the fact that our algorithms are deterministic in this proof. In fact, a similar argument can be applied to randomised algorithms. This is due to the fact that, as

we will see later, we consider problems that are equally hard for randomised and deterministic algorithms. Also, on truncated link machine encoding graphs the only way to cheat with error pointers is to produce a 2-colouring or copy the input of nodes that are far on the graph, that is, to solve problems that are equally hard for randomised and deterministic algorithms.

5.3 Instantiating the LCL Construction

We consider the problems Π_g and Π_l defined on cycles as follows:

Π_l (*3-colouring*): Output a proper 3-colouring.

 The time complexity of this problem is $\Theta(\log^* n)$ [8, 21], and it can be solved with one-sided algorithms. This is also clearly an LCL problem.

Π_g (*safe 2-colouring*): Given an input in $\{0, 1\}$, label the nodes with $\{0, 1, 2, \mathsf{E}\}$ such that

- input-0 nodes are labelled with 0,
- input-1 nodes are labelled with 1, 2, or E,
- 1 is never adjacent to 1,
- 2 is never adjacent to 2,
- E is never adjacent to 0, 1, or 2.

 In essence, if we have an all-1 input, we can produce an all-E output, and if we have an all-0 input, we can produce an all-0 output. However, if we have a mixture of 0s and 1s, we must properly 2-colour each contiguous chain of 1s. The worst-case instance is a cycle with only one 0, in which case we must properly 2-colour a chain of length $n - 1$.

 The time complexity of this problem is $\Theta(n)$, and it can be solved with one-sided algorithms. It is also clearly an LCL problem. Note that, unlike 2-colouring, safe 2-colouring is always solvable for any input (including odd cycles).

We now instantiate our LCL construction using the link machines defined in Section 3. The general recipe of these instantiations will be the following:

- We start with a link machine program M with growth g, and compute the function $f(\ell) = \ell g(\ell)$ that controls the speed-up.
- Next, we observe that there can be link machine encoding graphs with n nodes, in which the bottom cycle has length $\hat{n}$ satisfying $n = \Theta(\hat{n} \cdot f^{-1}(T(\hat{n})))$, in which nodes of the bottom cycle see no errors within distance $\Theta(f^{-1}(T(\hat{n})))$.
- Now it follows from Theorems 3 and 4 that when we instantiate the construction, we get problems of complexity
  (1) $T_1(n) = \Theta(f^{-1}(\hat{n}))$ when starting from Π_g, and
  (2) $T_2(n) = \Theta(f^{-1}(\log^* \hat{n}))$ when starting from Π_l.

By considering each of the composite functions of Table 4, and by applying Theorems 3 and 4, we obtain all of the new time complexities listed in Table 2.

Theorem 9. *There exist* LCL *problems of complexities*

 (1) $\Theta(n^{r/s})$,
 (2) $\Theta((\log^* n)^{q/p})$,

where r, s, p and q are positive integer constants, satisfying $q/p \leq 1$ and $r/s < 1/2$.

 PROOF. Let $M = \mathrm{POW}_p \circ \mathrm{ROOT}'_q$ with growth $g(\ell) = \Theta(\ell^{p/q})$. We have

- $f(\ell) = \Theta(\ell^{(p+q)/q})$, and
- $f^{-1}(x) = \Theta(x^{q/(p+q)})$.

When Π is Π_g we obtain:

- $n = \Theta(\hat{n}^{(p+2q)/(p+q)})$
- $\hat{n} = \Theta(n^{(p+q)/(p+2q)})$
- $T_1(n) = \Theta(\hat{n}^{q/(p+q)}) = \Theta(n^{q/(p+2q)})$.

By setting $q = r$ and $p = s - 2r$ the claim follows.
When Π is Π_l we obtain:

- $\hat{n} = \widetilde{\Theta}(n)$
- $T_2(n) = \Theta((\log^* n)^{q/(p+q)})$.

The claim follows by setting the values of p and q appropriately. □

Theorem 10. *There exist* LCL *problems of complexities*

 (1) $\Theta(\log^{p/q} n)$, *and*
 (2) $\Theta(\log^{p/q} \log^* n)$,

where p and q are positive integer constants such that $p/q \geq 1$.

 PROOF. Let $M = \mathrm{EXP} \circ \mathrm{POW}_q \circ \mathrm{ROOT}'_p$ with growth $g(\ell) = 2^{\Theta(\ell^{q/p})}$. We have

- $f(\ell) = 2^{\Theta(\ell^{q/p})}$ and $f^{-1}(x) = \Theta(\log^{p/q} x)$,
- $\hat{n} = \widetilde{\Theta}(n)$.

Thus, the LCL problem Π_M has complexity

- $T_1(n) = \Theta(\log^{p/q} \hat{n}) = \Theta(\log^{p/q} n)$ when Π is Π_g, and
- $T_2(n) = \Theta(\log^{p/q} \log^* \hat{n}) = \Theta(\log^{p/q} \log^* n)$ when Π is Π_l.
  □

Theorem 11. *There exist* LCL *problems of complexities*

 (1) $2^{\Theta(\log^{q/p} n)}$, *and*
 (2) $2^{\Theta(\log^{q/p} \log^* n)}$,

where p and q are positive integer constants such that $q/p \leq 1$.

 PROOF. Let $M = \mathrm{EXP} \circ \mathrm{POW}_p \circ \mathrm{ROOT}_q \circ \mathrm{LOG} \circ \mathrm{COUNT}$ with growth $g(\ell) = 2^{\Theta(\log^{p/q} \ell)}$. We have

- $f(\ell) = 2^{\Theta(\log^{p/q} \ell)}$ and $f^{-1}(x) = 2^{\Theta(\log^{q/p} x)}$,
- $\hat{n}$ is $\Omega(n^{1/2})$ and $O(n)$.

Thus, the LCL problem Π_M has complexity

- $T_1(n) = 2^{\Theta(\log^{q/p} \hat{n})} = 2^{\Theta(\log^{q/p} n)}$ when Π is Π_g, and
- $T_2(n) = 2^{\Theta(\log^{q/p} \log^* \hat{n})} = 2^{\Theta(\log^{q/p} \log^* n)}$ when Π is Π_l. □

REFERENCES

[1] Leonid Barenboim. 2016. Deterministic ($\Delta + 1$)-coloring in sublinear (in Δ) time in static, dynamic, and faulty networks. *J. ACM* 63, 5 (2016), 47:1–47:22. https://doi.org/10.1145/2979675

[2] Leonid Barenboim, Michael Elkin, and Fabian Kuhn. 2014. Distributed ($\Delta + 1$)-coloring in linear (in Δ) time. *SIAM J. Comput.* 43, 1 (2014), 72–95. https://doi.org/10.1137/12088848X

[3] Sebastian Brandt, Orr Fischer, Juho Hirvonen, Barbara Keller, Tuomo Lempiäinen, Joel Rybicki, Jukka Suomela, and Jara Uitto. 2016. A lower bound for the distributed Lovász local lemma. In *Proc. 48th Annual ACM Symposium on the Theory of Computing (STOC 2016)*. ACM, 479–488. https://doi.org/10.1145/2897518.2897570 arXiv:1511.00900.

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

[5] Yi-Jun Chang, Qizheng He, Wenzheng Li, Seth Pettie, and Jara Uitto. 2018. The complexity of distributed edge colouring with small palettes. In *Proc. 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2018)*. SIAM, 2633–2652. https://doi.org/10.1137/1.9781611975031.168

[6] Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. 2016. 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)*. IEEE, 615–624. arXiv:1602.08166.

[7] Yi-Jun Chang and Seth Pettie. 2017. A time hierarchy theorem for the LOCAL model. In *Proc. 58th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2017)*. IEEE, 156–167. https://doi.org/10.1109/FOCS.2017.23 arXiv:1704.06297.

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

[9] Michael Elkin, Seth Pettie, and Hsin-Hao Su. 2015. $(2\Delta-1)$-edge-coloring is much easier than maximal matching in the distributed setting. In *Proc. 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2015)*. SIAM, 355–370. https://doi.org/10.1137/1.9781611973730.26

[10] Manuela Fischer. 2017. Improved deterministic distributed matching via rounding. In *Proc. 31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl – Leibniz Center for Informatics, 17:1–17:15. https://doi.org/10.4230/LIPIcs.DISC.2017.17

[11] Manuela Fischer and Mohsen Ghaffari. 2017. Sublogarithmic distributed algorithms for Lovász local lemma, and the complexity hierarchy. In *Proc. 31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl – Leibniz Center for Informatics, 18:1–18:16. https://doi.org/10.4230/LIPIcs.DISC.2017.18

[12] Manuela Fischer, Mohsen Ghaffari, and Fabian Kuhn. 2017. Deterministic distributed edge-coloring via hypergraph maximal matching. In *Proc. 58th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2017)*. IEEE, 180–191. https://doi.org/10.1109/FOCS.2017.25

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

[14] Mohsen Ghaffari. 2016. An improved distributed algorithm for maximal independent set. In *Proc. 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2016)*. SIAM, 270–277. https://doi.org/10.1137/1.9781611974331.ch20

[15] Mohsen Ghaffari, Juho Hirvonen, Fabian Kuhn, Yannic Maus, Jukka Suomela, and Jara Uitto. 2017. Improved distributed degree splitting and edge coloring. In *Proc. 31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl – Leibniz Center for Informatics, 19:1–19:15. https://doi.org/10.4230/LIPIcs.DISC.2017.19

[16] Mohsen Ghaffari and Hsin-Hao Su. 2017. Distributed degree splitting, edge coloring, and orientations. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*. SIAM, 2505–2523. https://doi.org/10.1137/1.9781611974782.166

[17] Mika Göös and Jukka Suomela. 2016. Locally checkable proofs in distributed computing. *Theory of Computing* 12, 19 (2016), 1–33. https://doi.org/doi:10.4086/toc.2016.v012a019

[18] David G. Harris, Johannes Schneider, and Hsin-Hao Su. 2016. Distributed $(\Delta+1)$-coloring in sublogarithmic rounds. In *Proc. 48th Annual ACM Symposium on the Theory of Computing (STOC 2016)*. ACM, 465–478. https://doi.org/10.1145/2897518.2897533

[19] Amos Korman, Shay Kutten, and David Peleg. 2010. Proof labeling schemes. *Distributed Computing* 22, 4 (2010), 215–233. https://doi.org/10.1007/s00446-010-0095-3

[20] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. 2016. Local computation: lower and upper bounds. *J. ACM* 63, 2 (2016), 17:1–17:44. https://doi.org/10.1145/2742012 arXiv:1011.5470.

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

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

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

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

[25] Alessandro Panconesi and Aravind Srinivasan. 1996. On the complexity of distributed network decomposition. *Journal of Algorithms* 20, 2 (1996), 356–374. https://doi.org/10.1006/jagm.1996.0017

[26] David Peleg. 2000. *Distributed Computing: A Locality-Sensitive Approach*. SIAM.