Constant Space and Non-Constant Time in Distributed Computing

Tuomo Lempiäinen[1] **and Jukka Suomela**[2]

1 **Department of Computer Science, Aalto University, Espoo, Finland**
tuomo.lempiainen@aalto.fi

2 **Department of Computer Science, Aalto University, Espoo, Finland**
jukka.suomela@aalto.fi

Abstract

While the relationship of time and space is an established topic in traditional centralised complexity theory, this is not the case in distributed computing. We aim to remedy this by studying the time and space complexity of algorithms in a weak message-passing model of distributed computing. While a constant number of communication rounds implies a constant number of states visited during the execution, the other direction is not clear at all. We show that indeed, there exist non-trivial graph problems that are solvable by constant-space algorithms but that require a non-constant running time. Somewhat surprisingly, this holds even when restricted to the class of only cycle and path graphs. Our work provides us with a new complexity class for distributed computing and raises interesting questions about the existence of further combinations of time and space complexity.

1998 ACM Subject Classification F.1.1 Models of Computation, F.1.3 Complexity Measures and Classes

Keywords and phrases distributed computing, space complexity, constant-space algorithms, weak models, Thue–Morse sequence

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.30

1 Introduction

In the classical centralised theory of computing, the study of space-limited computation has helped us with understanding computability and computational complexity in general:

1. Constant-space models (finite-state machines) provide a very well-understood solid foundation.
2. Space-limited complexity classes (e.g., PSPACE) can be successfully related with time-limited complexity classes (e.g., NP $\subseteq$ PSPACE $\subseteq$ EXP).

In this work, we use similar ideas in the study of distributed computing, in particular, in the context of distributed graph algorithms.

Networks of finite-state machines. The natural distributed analogue of a deterministic finite-state machine is a *network* of deterministic finite-state machines. For brevity, we call distributed algorithms that use only finitely many states per node *constant-space algorithms*. See Section 2 for a proper definition of the model of computation.

Variants of this setting have been studied in many papers, but perhaps the most elementary question related to distributed graph algorithms has not been answered yet:

Can we say anything about the *time* complexity of graph problems that are solvable with a network of finite-state machines?

Naturally, many constant-time algorithms are also constant-space algorithms, but does the converse hold in general?

Key challenges. At first, the question may seem trivial: a path of n finite-state machines can simulate the computation of any Turing machine with n units of space. We can easily construct algorithms that take time that is exponential in n to terminate; a simple example is a binary counter that counts from all-0 to all-1.

However, this does not imply that there are graph problems that are solvable in constant space but not in constant time! There are two main obstacles:

1. The fact that there exists a slow algorithm does not imply that the problem cannot be solved with a fast algorithm. In the binary counter example, we produce an all-1 output if we have an all-0 input; this is of course a trivial problem to solve in constant time.
2. We exploit a *promise*: we assume that the input is a path.

The second obstacle may at first seem like a mere technicality, but it turns out to be the key difference that separates the centralised computational complexity theory from its distributed counterpart when we consider constant-space machines.

In centralised computing, the input is by definition a string with a well-defined starting point and endpoint. A natural analogue of this in distributed computing is a (directed) path. If we have a promise that the input is a path, we can easily construct examples of problems that are solvable with constant-space but not with constant-time algorithms, by following ideas that work in the case of centralised computing:

- Classical centralised language: "Recognise the language of strings that have an even number of characters."
- Distributed graph problem: "If the input is a path with an even number of nodes, all nodes have to output 1, otherwise all nodes have to output 0."

However, in distributed graph algorithms, the input is a graph that comes from an adversary (see Section 2.4), and the graph may be highly symmetric. A prime example is a cycle graph; indeed, breaking symmetry in cycles is one of the most classical problems in the area of distributed graph algorithms.

Informally, centralised finite-state machines never get stuck in infinite loops, as eventually the input string that they are scanning will end, but a distributed algorithm might easily fail to terminate if the input is a cycle. The usual escape is to assume that we have globally unique node identifiers (or access to randomness), but non-constant-size identifiers are not a natural assumption with constant-space algorithms.

Indeed, if we insist that the algorithm is a well-defined algorithm in the sense that it always terminates, it is difficult to see if we can solve *anything* non-trivial with constant-space algorithms without resorting to some assumption on the input: all nodes have to terminate in some finite time T even if the input is a cycle, and a cycle is indistinguishable from a sufficiently long path, and hence all nodes have to terminate in fixed time T in arbitrarily long paths. It would be tempting to conjecture that constant space (and well-defined algorithms without any promise) would necessarily imply constant time. In this work, we show that this is not the case.

Contributions. We give a counterexample that separates constant-space and constant-time algorithms (see Section 4). More precisely, we construct a well-defined graph problem Π such that:

- Π is solvable with constant-space algorithms (even if we consider a very restricted subset of constant-space machines),
- Π cannot be solved in $o(n)$ time with any distributed algorithm (even if we use a much stronger model of computation, e.g., randomised algorithms in the LOCAL model).

Here Π is a well-defined input–output problem in the classical sense: for every labelled input graph G we have a set $\Pi(G)$ of feasible output labellings. We only need constant-size input and output labels, and our result holds even if we restrict to input graphs of maximum degree at most 2.

The key technical challenge here is to engineer a problem Π and a constant-space algorithm $\mathcal{A}$ for Π such that for *any* input graph G, with *any* input labelling, algorithm $\mathcal{A}$ is guaranteed to stop in finite time; yet Π has to be sufficiently non-trivial so that it cannot be solved in constant time – not even in graphs of maximum degree 2. Here we resort to so-called Thue–Morse sequence as a source of inspiration (see Section 2.5).

Open questions. We emphasise that our problem Π is a purely artificial problem that only serves the purpose of separating constant-space and constant-time algorithms. Furthermore, many problems that have been extensively studied in the area of distributed graph algorithms are LCL (locally checkable labelling) problems [20], and our problem Π is not a member of this problem family. Hence there are immediate follow-up questions that we leave for future work:

1. Does there exist a *natural* graph problem that separates constant space and constant time?
2. Does there exist an LCL problem that separates constant space and constant time?

Related work. While the relationship between space and time complexity in distributed computing is a rather novel topic, various time complexity classes and the relationships between them have been studied a lot in many of the established works of the field [14, 16, 19, 20] and also very recently [3, 4, 5, 6, 7, 13].

The line of research on anonymous models of distributed computing was initiated by Angluin [2], who introduced the well-known port numbering model. Our model of computation can be seen as a further restriction of the port-numbering model, with port numbers stripped out. While port numbers are a natural assumption in wired networks, the weaker variant makes more sense when applying distributed computing to a wireless setting. While our model has not been studied that much in prior work, a so-called *beeping model* [1, 9] is essentially similar. In the hierarchy of seven models defined by Hella et al. [15], our model is the weakest one; they call it the SB model. On the other hand, the case where nodes receive messages in a multiset instead of a set is discussed more often in the literature [15].

From the constant-space point of view, our setting bears similarities to the field of cellular automata [12, 23, 24]. In a cellular automaton, each cell can be in one of a constant number of states, and each cell updates its state synchronously using the same rule. However, in the case of cellular automata, one is usually interested in the kind of patterns an automata converges in, while we require each node to eventually stop and produce an output.

On the side of distributed computing, Emek and Wattenhofer [11] have considered a model where the network consists of finite-state machines – hence making the space complexity constant. However, their model is asynchronous and randomised, while we study a fully synchronous and deterministic setting. Recently, also Kuusisto and Reiter [18, 21, 22] have considered networks of finite-state machines, but their machines either halt in constant time [21] or continue running indefinitely [18, 22]. Furthermore, constant memory has been studied in the setting where a set of mobile agents explores a graph [8, 10].

One of the main ingredients of our work, the Thue–Morse sequence, was used previously by Kuusisto [17], who proved that there exists a distributed algorithm that always halts in the class of graphs of maximum degree two but features a non-constant running time. However, his algorithm has also a non-constant space complexity – this is where our work provides a significant improvement.

2 Preliminaries

In this section, we define our model of computation and introduce notions needed later in the proofs.

2.1 Model of computation

We consider a model of computation where each node of a graph is a computational unit. The same graph is a communication network and an input to the algorithm. Adjacent nodes communicate with each other in synchronous rounds, and eventually each node outputs its own part of the output. All the nodes run the same deterministic algorithm. In our model, the nodes are anonymous, that is, they do not have access to unique identifiers, and furthermore, they cannot distinguish between their neighbours – they broadcast the same message to everyone and they receive the incoming messages in a set. Our model can be seen as a very weak variant of the standard LOCAL model of distributed computing.

Running time is defined as the number of communication rounds until all the nodes have halted, while space usage is defined as the number of bits per node needed to represent all the states of the algorithm. The complexity measures are considered as a function of n, the number of nodes in the graph. The amount of local computation in each round is not limited, nor is the size of messages (but constant space complexity implies that they are also constant).

In the following, we define distributed algorithms as state machines. Given an input graph, each node of the graph is equipped with an identical state machine. The graph is always assumed to be simple, finite, connected and undirected, unless stated otherwise.

At the beginning, each state machine is only aware of its own local input (taken from some fixed finite set) and the degree of the node on which it sits. Then, computation is executed in synchronous rounds. In each round, each machine

1. broadcasts a message to its neighbours,
2. receives a set of messages from its neighbours,
3. moves to a new state based on the received messages and its previous state.

Each machine is required to eventually reach one of special halting states and stop execution. The local output is then the state of the node at the time of halting.

Note that while our model of computation is rather weak, it only makes our results stronger by limiting the capabilities of algorithms. Unique identifiers or unrestricted local inputs would not make much sense in the constant-space setting. Crucially, we require nodes to always halt in any input graph. Engineering constant-space non-constant-time problems would be quite straightforward – even in the case of degree 2 – if nodes were allowed to run indefinitely or had the ability to continue passing messages after announcing an output.[1]

Similarly, randomisation would weaken the results: with an unlimited supply of random bits, one can acquire unique identifiers with a high probability, and even a single random bit per node would allow us to construct a simple constant-space algorithm that halts with a high probability in all paths and cycles, but exhibits a non-constant running time in the worst case.

Next, we give a more formal definition of the model of computation used in this work by defining algorithms and graph problems.

2.2 Notation and terminology

For $k \in \mathbb{N}$, we denote by $[k]$ the set $\{1, 2, \ldots, k\}$. Given a graph $G = (V, E)$, the set of neighbours of a node $v \in V$ is denoted by $N(v) = \{u \in V : \{v, u\} \in E\}$. For $r \in \mathbb{N}$, the *radius-r neighbourhood* of a node $v \in V$ is $\{u \in V : \operatorname{dist}(u, v) \le r\}$, that is, the set of nodes u such that there is a path of length at most r between v and u. We call any induced subgraph of G that contains node v simply a *neighbourhood* of v.

We will also work with strings of letters. Given a finite alphabet Σ, we denote elements of the alphabet by lowercase symbols such as $x \in \Sigma$ or $y \in \Sigma$. On the other hand, *words* (finite sequences of letters) are denoted by uppercase symbols, e.g., $X = x_1 x_2 \ldots x_i \in \Sigma^*$. Given words $X = x_1 x_2 \ldots x_i$ and $Y = y_1 y_2 \ldots y_j$, we write their concatenation simply $XY = x_1 x_2 \ldots x_i y_1 y_2 \ldots y_j$. A word X is a *subword* of Y if $Y = Y_1 X Y_2$ for some (possibly empty) words Y_1 and Y_2. We identify words of length 1 with the letter they consist of. For any letter x and $i \in \mathbb{N}$, x^i denotes the word consisting of i consecutive letters x. We say that a word X is *of the form* $x+$ if $X = x^i$ for some $i \in \mathbb{N}_+$.

2.3 Algorithms as state machines

Let $G = (V, E)$ be a graph. An *input* for G is a function $f \colon V \to I$, where I is a finite set. For each node $v \in V$, we call $f(v) \in I$ the *local input* of v.

A *distributed state machine* is a tuple $\mathcal{A} = (S, H, \sigma_0, M, \mu, \sigma)$, where

- S is a set of states,
- $H \subseteq S$ is a finite set of halting states,
- $\sigma_0 \colon \mathbb{N} \times I \to S$ is an initialisation function,
- M is a set of possible messages,
- $\mu \colon S \to M$ is a function that constructs the outgoing messages,
- $\sigma \colon S \times \mathcal{P}(M) \to S$ is a function that defines the state transitions, so that $\sigma(h, \mathcal{M}) = h$ for each $h \in H$ and $\mathcal{M} \in \mathcal{P}(M)$.

Given a graph G, and input f for G and a distributed state machine $\mathcal{A}$, the *execution* of $\mathcal{A}$ on (G, f) is defined as follows. The state of the system in round $r \in \mathbb{N}$ is a function $x_r \colon V \to S$, where $x_r(v)$ is the state of node v in round r. To begin the execution, set $x_0(v) = \sigma_o(\deg(v), f(v))$ for each node $v \in V$. Then, let $A_{r+1}(v) = \{\mu(x_r(u)) : u \in N(v)\}$

[1] Consider the following problem: nodes with local input 0 always output 0, while nodes with local input 1 output the shortest distance modulo 2 to either a degree-1 node or to a node (possibly the node itself in the case of a cycle) with local input 1. Now, nodes with local input 0 can announce their output immediately (regardless of the existence of nodes with input 1), while nodes with local input 1 have to potentially wait for a linear number of rounds.

denote the set of messages received by node v in round $r+1$. Now the new state of each node $v \in V$ is defined by setting $x_{r+1}(v) = \sigma(x_r(v), A_{r+1}(v))$.

The *running time* of $\mathcal{A}$ on (G, f) is the smallest $t \in \mathbb{N}$ for which $x_t(v) \in H$ holds for all $v \in V$. The output of $\mathcal{A}$ on (G, f) is then $x_t \colon V \to H$, where t is the running time, and for each $v \in V$, the *local output* of v is $x_t(v)$. The *space usage* of $\mathcal{A}$ on (G, f) is defined as

$$\lceil \log |\{s \in S \,:\, s = x_r(v) \text{ for some } r \in \mathbb{N}, v \in V\}| \rceil,$$

that is, the number of bits needed to encode all the states that the state machine visits in at least one node of G during the execution. In case the execution does not halt, the running time can be defined to be ∞, and if the number of visited states grows arbitrarily large, we take the space usage to also be ∞.

From now on, we will use the terms *algorithm* $\mathcal{A}$ and *distributed state machine* $\mathcal{A}$ interchangeably, implying that each algorithm can be defined formally as a state machine.

2.4 Graph problems

The computational problems that we consider are graph problems with local input – that is, the problem instance is identical to the communication network, but possibly with an additional input value given to each node.

More formally, let I and O be finite sets. A *graph problem* is a mapping $\Pi_{I,O}$ that maps each graph $G = (V, E)$ and input $f \colon V \to I$ to a set $\Pi_{I,O}(G, f)$ of valid solutions. Each solution S is a function $S \colon V \to O$. In case of *decision graph problems*, we set $O = \{\mathsf{yes}, \mathsf{no}\}$.

If $\Pi_{I,O}$ is a graph problem, $T, U \colon \mathbb{N} \to \mathbb{N}$ are functions, $\mathcal{A}$ is a distributed state machine and $\mathcal{G}$ is a class of graphs, we say that $\mathcal{A}$ *solves* $\Pi_{I,O}$ *in class* $\mathcal{G}$ *in time* T *and in space* U if for each graph $G = (V, E) \in \mathcal{G}$ and each input $f \colon V \to I$ we have that the running time of $\mathcal{A}$ on (G, f) is at most $T(|V|)$, the space usage of $\mathcal{A}$ on (G, f) is at most $U(|V|)$ and the output of $\mathcal{A}$ on (G, f) is in the set $\Pi_{I,O}(G, f)$. In that case, we also say that the *time complexity* of algorithm $\mathcal{A}$ in $\mathcal{G}$ is T and the *space complexity* of $\mathcal{A}$ in $\mathcal{G}$ is U. If $\mathcal{G}$ is omitted, it is assumed to be the class of all (simple, finite, connected and undirected) graphs.

We define the *time complexity* of a problem $\Pi_{I,O}$ in a graph class $\mathcal{G}$ to be the slowest-growing function $T \colon \mathbb{N} \to \mathbb{N}$ such that there exists an algorithm $\mathcal{A}$ that solves $\Pi_{I,O}$ in $\mathcal{G}$ in time T. The *space complexity* of a problem is defined analogously. Again, we omit $\mathcal{G}$ when it is the class of all graphs. Notice that when considering a lower bound for the complexity of a problem, the smaller the class $\mathcal{G}$, the stronger the result, while for upper bounds, the situation is reverse.

Some typical examples of graph problems considered in the distributed setting are the minimum vertex cover problem and the maximal independent set problem, where for each valid solution S, $S(v) = 1$ indicates that node v is part of the vertex cover or independent set, respectively, while $S(v) = 0$ indicates that it is not part of it. In the setting used in this work, where nodes are allowed to receive local inputs, some natural examples are the problem of outputting the largest input value among each node's neighbours, or the problem of verifying that a colouring given as an input is proper.

2.5 Thue–Morse sequence

In this section, we present a concept that will be central in the proof of our main result in Section 4. The Thue–Morse sequence is the infinite binary sequence defined recursively as follows:

▶ **Definition 1.** The *Thue–Morse* sequence is the sequence (t_i) satisfying $t_0 = 0$, and for each $i \in \mathbb{N}$, $t_{2i} = t_i$ and $t_{2i+1} = 1 - t_i$.

Thus, the beginning of the Thue–Morse sequence is

$$0110100110010110 1001\ldots$$

For our purposes, the following two recursive definitions will be very useful.

▶ **Definition 2.** Let $T_0 = 0$. For each $i \in \mathbb{N}_+$, let $T_i = T_{i-1}\mathcal{C}(T_{i-1})$, where $\mathcal{C}$ denotes the Boolean complement.

Note that for each $i \in \mathbb{N}$, the word T_i is the prefix of length 2^i of the Thue–Morse sequence.

▶ **Definition 3.** Let $T'_0 = 0$. For each $i \in \mathbb{N}_+$, let T'_i be obtained from T'_{i-1} by substituting each occurrence of 0 with 01 and each occurrence of 1 with 10.

Again, we have $T'_0 = 0$, $T'_1 = 01$, $T'_2 = 0110$, $T'_3 = 01101001$ and so on. A straightforward induction shows that the above two definitions are equivalent: $T_i = T'_i$ for each $i \in \mathbb{N}$. We call T_i the *Thue–Morse word of length* 2^i.

The Thue–Morse sequence contains lots of squares, that is, subwords of the form XX, where $X \in \{0, 1\}^*$. Interestingly, it does not contain any cubes – subwords of the form XXX. Note also that for each $i \in \mathbb{N}_+$, T_{2i} is a palindrome.

3 Warm-up: graphs of maximum degree 3

In this section, we present a graph problem that exhibits the combination of constant space complexity and non-constant time complexity, in case we do not restrict ourselves to paths and cycles. The proof is quite straightforward, which emphasises the fact that the degree-2 case considered in Section 4 is the most interesting one. We emphasise that in the following theorem, we do not need to make any additional assumptions about the graph; the described algorithm halts in all finite input graphs.

▶ **Theorem 4.** *There exist a graph problem* Π *with constant space complexity and* $\Theta(\log n)$ *time complexity. Furthermore, the time complexity of* Π *is* $\Theta(\log n)$ *also in the class of graphs of maximum degree at most 3.*

Proof. Consider the following transformation: given a *directed* graph $G' = (V', E')$ with edges labelled "L" or "R", replace each edge $e = (u, v)$ by a gadget as represented in Figure 1. The end result is an undirected graph $G = (V, E)$, where the gadgets encode the edge directions and labels of the original graph. We say that an undirected graph G is *good* if it is obtained by the above transformation from some directed binary pseudotree G' (that is, a connected directed graph where each node has outdegree at most one and indegree either 0 or 2) such that each node with incoming edges has one incoming edge labelled with "L" and one with "R".

We call a leaf node of a good graph *original*, if it does not belong to any gadget. Now, consider the following graph problem: if the input graph is good, each node has to output the parity of its distance to the nearest original leaf node; if not, each node is allowed to output anything. In any case, all the nodes have to halt.

Define an algorithm $\mathcal{A}$ as follows. First, $\mathcal{A}$ checks locally in two phases if the input graph G is good. In the first phase, each node detects in five communication rounds whether it is contained in a valid gadget or neighboured by valid gadgets. If so, then in the second phase each node v corresponding to a node v' from the original directed graph checks that in the orientation obtained from the neighbouring gadgets, the node v' either has

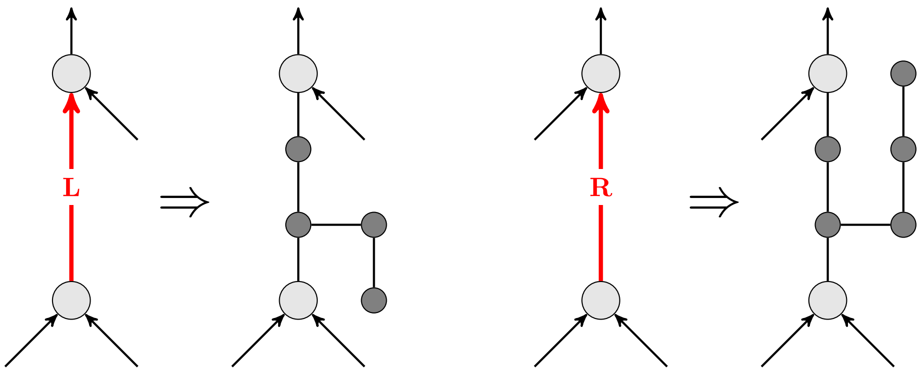

Figure 1 The red edge is replaced by the dark grey gadget consisting of four or five nodes, depending on the label. Similar encoding is performed to each edge of the original directed graph.

- two incoming edges (labelled "L" and "R") and one outgoing edge,
- only one outgoing edge (it is a leaf node), or
- only two incoming edges (it is the root node in case of a tree).

If either of the checks fails, the node broadcasts an instruction to halt to its neighbours and halts. Otherwise, it follows that the original graph G' is indeed a directed binary pseudotree with each edge directed away from the leaf nodes.

After verifying the goodness of the instance, algorithm $\mathcal{A}$ starts to count the distances to original leaf nodes. In the first round, each original leaf node broadcasts message 0 to its neighbours, halts and outputs 0. When a node that has not yet halted receives message i, where $i \in \{0, 1\}$, it broadcasts $i + 1 \bmod 2$ to its neighbours, halts and outputs $i + 1 \bmod 2$. Because the input graph is assumed to be finite and connected, each node eventually halts.

Since verifying the goodness takes only a constant number of rounds, and in the counting phase we count only up to 2, algorithm $\mathcal{A}$ needs only a constant number of states. Furthermore, as in any binary pseudotree the distance from any node to the closest leaf node is at most $O(\log n)$, each node halts after at most $O(\log n)$ rounds. On the other hand, in a balanced binary tree, the distance from the root to leaf nodes is $\Omega(\log n)$, and thus $\Omega(\log n)$ rounds are needed until all the nodes can output the parity of their distance to the closest original leaf. This completes the proof. ◀

4 Graphs of maximum degree 2

We are now ready to state the main theorem of this work.

▶ **Theorem 5.** *There exist a decision graph problem Π with constant space complexity and $\Theta(n)$ time complexity. Furthermore, the time complexity of Π is $\Theta(n)$ also in the class of graphs of maximum degree at most 2.*

To define our graph problem, we will make use of the Thue–Morse sequence via the following definition – compare this to Definition 3 earlier.

▶ **Definition 6** (Valid words). Define a set of words over $\{0, 1, \_\}$ recursively as follows:

1. _0_ is *valid*,
2. if X is valid and Y is obtained from X by applying the substitutions $0 \mapsto$ 0_1_1_0 and $1 \mapsto$ 1_0_0_1 to each occurrence of 0 and 1, then Y is *valid*.

That is, valid words are obtained from the sequences $T_{2i}, i \in \mathbb{N}$, by inserting an underscore at the beginning, in between each symbol and at the end.

Now, we will define a decision graph problem that we call ThueMorse as follows.

▶ **Definition 7** (ThueMorse)**.** The local inputs of the problem are taken from the set $I = \{\alpha, \beta, \gamma\} \times \{0, 1, \_\}$ and each local output is either yes or no. Given an input function $f\colon V \to I$, we write $f = (f_1, f_2)$. Now, an instance (G, f) is a yes-instance of ThueMorse if and only if

- the graph $G = (V, E)$ is a path graph,
- for each $v \in V$, we have

  $$\{f_1(u) \,:\, u = v \text{ or } u \in N(v)\} = \{\alpha, \beta, \gamma\},$$

  that is, by setting $\alpha < \beta < \gamma$, the first parts $f_1(\cdot)$ of the local inputs define a consistent orientation for the path,
- if we denote $V = \{v_1, v_2, \dots, v_n\}$ where $\{v_i, v_{i+1}\} \in E$ for each $i \in \{1, 2, \dots, n-1\}$, the word

  $$f_2(v_1)f_2(v_2)\dots f_2(v_n)$$

  defined by the second parts of the local inputs is valid.

4.1 Definition of the algorithm

Next, we will give an algorithm that is able to solve the decision problem ThueMorse by using only constant space. The high-level idea is as follows: First, we check that the local neighbourhood of each node looks correct. Then, we repeatedly apply substitutions that roll the configuration of the path graph back to a shorter prefix of the Thue–Morse sequence, until we reach a trivial configuration and accept the input – or fail to apply the substitutions unambiguously and consequently reject the input.

▶ **Lemma 8.** *There exists an algorithm $\mathcal{A}$ that solves problem* ThueMorse *in space $O(1)$ and time $O(n)$.*

First, let us introduce some terminology and notation. In an instance (G, f) of ThueMorse where G is either a path or a cycle graph, the sequence of symbols defined by f_2 in a given node neighbourhood is called the *input word* – note that contrary to usual words, the input word is unoriented. Sometimes we identify the node neighbourhood with the corresponding input word; the meaning should be clear from the context. During the execution of the algorithm, each node v has a *current symbol* $c(v)$ as part of its state. The vertical bar $|$ shall denote the end of the path graph. We will make use of it by assuming that each degree-1 node v sees a "virtual" neighbour u with the symbol $|$, that is, $f_1(u) = f_2(u) = |$ and $c(u)$ is always equal to $|$. The underscore symbols _ will be called *separators*. Denote the alphabet which contains the possible current symbols by $\Sigma = \{0, 1, \_, |\}$.

We define the algorithm $\mathcal{A}$ in three parts, which we denote by I, II and III. In each part, any node can *abort*, which means that it sends a special abort message to its neighbours and then moves to state no and thus halts. If a node receives the abort message at any time, it aborts – that is, it passes the message on to all its neighbours, and then moves to state no and halts. At initialisation, each node v sets its symbol $c(v)$ to be equal to $f_2(v)$, the second part of the local input.

In **part I**, each node v verifies its degree and the orientation: if $\deg(v) \in \{1, 2\}$ and three different symbols from $\{\alpha, \beta, \gamma, |\}$ can be found in the local inputs within the radius-1 neighbourhood of v (that is, we have $|\{f_1(u) : u \in N(v) \cup \{v\}\}| = 3$), continue; otherwise, abort. Recall that the graph is assumed to be finite and connected. If none of the nodes aborts, it follows that the graph is either a path or a cycle and that the local inputs given by f_1 define a word of the form $\dots\alpha\beta\gamma\alpha\beta\gamma\alpha\beta\gamma\alpha\dots$.

In **part II**, each node v verifies the input word in its radius-1 neighbourhood: if the neighbourhood is in {|_0, 0_|, 0_0, 1_1, 0_1, 1_0, _0_, _1_}, continue; otherwise, abort. Note that here we do not care about the orientation of the word, and hence the set of symbols received from the neighbours is enough for this step. If none of the nodes aborts, the input word is *locally valid*: every other symbol is the separator _ and every other symbol is either 0 or 1.

Then, we proceed to **part III**, which contains the most interesting steps of the algorithm. Now we will make use of the orientation given as part of the input. Define $\alpha < \beta$, $\beta < \gamma$ and $\gamma < \alpha$. For each node v, we say that neighbour u of v is the *left neighbour* of v if $f_1(u) < f_1(v)$ and the *right neighbour* of v if $f_1(u) > f_1(v)$. Thus, each node can essentially send a different message to each of its two neighbours: the orientation symbol of the recipient indicates which part of the message is intended for which recipient. From now on, we will also assume that each node attaches its own orientation symbol as part of every message sent, so that nodes can distinguish between incoming messages.

Part III will consist of several *phases*. In each phase, we first gather information from the neighbourhood in two *buffers* and then try to apply a substitution to the word obtained in the buffers. More precisely, each node v has two buffers, the left buffer $L(v)$ and the right buffer $R(v)$ as part of its state. The buffers are used to store a *compressed* version of the input word in the neighbourhood: a sequence of consecutive symbols 0 or 1 is represented by a single 0 or 1, respectively.

Let us define some notation. Let $l, r\colon \Sigma^* \times \Sigma \to \Sigma^*$ be as follows. If $X = Ay$ for some word A, set $r(X, y) = X$. Otherwise, set $r(X, y) = Xy$. Function l is defined analogously: if $X = yA$ for some word A, set $l(X, y) = X$; otherwise, set $l(X, y) = yX$. In other words, functions l and r append a new symbol either to the beginning or to the end of a word, respectively – but only if the new symbol is different from the current first or last symbol of the word, respectively.

Now, in each phase, each node v first *fills its buffers* as follows. To initialise, node v broadcasts message $c(v)$ (its own current symbol) to its neighbours. Then, node v repeats the following. It sets the left buffer $L(v)$ equal to the message it receives from its left neighbour and the right buffer $R(v)$ equal to the message it receives from the right neighbour. After that, node v sends $r(L(v), c(v))$ to the right and $l(R(v), c(v))$ to the left. These steps are repeated until both buffers of v contain either 8 instances of the separator _ or an end-of-the-path marker |. When that happens, node v is finished with filling its buffers.

Next, node v combines its buffers to construct a *compressed view* of the input word in its neighbourhood. To that end, define $C\colon \Sigma^* \times \Sigma \times \Sigma^* \to \Sigma^*$ and $p\colon \Sigma^* \times \Sigma \times \Sigma^* \to \mathbb{N}$ as follows. If $X = Ay$ and $Z = yB$ for some $A, B \in \Sigma^*$, set $C(X, y, Z) = AyB$ and $p(X, y, Z) = |A|$. Otherwise, if $X = Ay$ for some $A \in \Sigma^*$, set $C(X, y, Z) = AyZ$ and $p(X, y, Z) = |A|$, and if $Z = yB$ for some $B \in \Sigma^*$, set $C(X, y, Z) = XyB$ and $p(X, y, Z) = |X|$. Else, set $C(X, y, Z) = XyZ$ and $p(X, y, Z) = |X|$. Now, the compressed view of node v is $V(v) = C(L(v), c(v), R(v))$, and $c(v)$ is at position $q(v) = p(L(v), c(v), R(v)) + 1$ in $V(v)$. In other words, the left buffer, the current symbol and the right buffer are concatenated in a way that removes successive repetitions of the same symbol.

Finally, node v does subword matching on the view $V(v)$. If $V(v)$ is equal to |_0_| or |_0_1_1_0_|, node v instructs other nodes to accept, moves to the yes state and halts.

Otherwise, node v searches $V(v)$ for the subword _0_1_1_0_1_0_0_1_ in all possible positions. Given a match, let the ith symbol of the subword be aligned with the $q(v)$th symbol of $V(v)$. If $i \in \{1, 9, 17\}$, set $c' = \_$. Otherwise, if $i \le 8$, set $c' = 0$, and if $i \ge 10$, set $c' = 1$. After that, node v performs the same procedure with the reversed subword _1_0_0_1_0_1_1_0_ – but now, if $i \le 8$, set $c' = 1$, and if $i \ge 10$, set $c' = 0$. If no matches could be found in $V(v)$, node v aborts. If several matches were found and they resulted in different values for c', node v aborts. Otherwise, node v updates its current symbol $c(v)$ to be the unambiguous value c'. This concludes the phase; if not aborted, on the next round, a new phase starts from the beginning. See Example 9 for illustrations of the matching and substitution steps in a few cases.

▶ **Example 9.** Consider the execution of part III of algorithm $\mathcal{A}$ in the following instances.

1. Path graph, yes-instance:

   |_0_1_1_0_1_0_0_1_1_0_0_1_0_1_1_0_|

   $\Downarrow$ (unambiguous substitutions)

   |_0000000_1111111_1111111_0000000_|

   $\Downarrow$ ($V(v) = $ |_0_1_1_0_|)

   accept

2. Path graph, no-instance:

   |_0_1_1_0_1_0_0_1_1_0_1_0_0_1_|

   $\Downarrow$

   |_0000000_1111111_...

   ..._0000000_1111111_|

   $\Downarrow$ (ambiguous substitutions)

   abort

3. Cycle graph (the ends marked with ... are connected circularly):

   ..._0_1_1_0_1_0_0_1_1_0_0_1_0_1_1_0_...

   $\Downarrow$ (unambiguous substitutions)

   ..._0000000_1111111_1111111_0000000_...

   $\Downarrow$ (no matches)

   abort

4.2 Proof of correctness

In this section, we show that algorithm $\mathcal{A}$ executes correctly and always halts with the desired output. Since parts I and II are quite trivial, we will start with part III of the algorithm.

We say that a word $X = x_1x_2 \dots x_i$ is the *compressed version* of a word $Y = y_1y_2 \dots y_j$ if $x_i \ne x_{i+1}$ for all $i \in \{1, 2, \dots, i-1\}$ and there exist a surjective *compression mapping* $f\colon [j] \to [i]$ such that $f(1) = 1$, $f(k) \in \{f(k-1), f(k-1)+1\}$ for all $k \in \{2, 3, \dots, j\}$ and $y_k = x_{f(k)}$ for all $k \in [j]$.

▶ **Lemma 10.** *After any node v has finished collecting the buffers, $V(v)$ is the compressed version of the actual input word in the neighbourhood of v.*

Proof. We use induction on the number of rounds t after starting the phase. After the first round of the phase, each node v has received $L(v) = c(u)$ from its left neighbour u and $R(v) = c(w)$ from its right neighbour w. Hence $L(v)$ and $R(v)$ are compressed versions of the left and right 1-neighbourhoods of v, respectively.

Assume then that $L(u)$ and $R(w)$ are compressed versions of the left and right $(t-1)$-neighbourhoods of u and w, respectively, and let f_l and f_r be the corresponding mappings. Consider now the definition of the algorithm. The definition of the mapping r implies that what v receives from the left in round t of the phase is $L(u)$ – extended by $c(u)$ if and only if $c(u)$ differs from the last symbol of $L(u)$. If it does differ, we extend f_l by defining $f_l(t) = |L(u)| + 1$, otherwise $f_l(t) = |L(u)|$. Hence the new value of the buffer $L(v)$ is a compressed version of the left t-neighbourhood of v. The case of $R(v)$ is handled analogously.

Suppose that v has finished collecting the buffers. Now $L(v)$ and $R(v)$ are the compressed versions of the left and right k-neighbourhoods of v for some k. Then, it follows from the definition of the function C that an appropriate compression mapping can be formed and $V(v) = C(L(v), c(v), R(v))$ is the compressed version of the k-neighbourhood of v. ◀

We call the sequence of current symbols $c(u)$ in the graph a *configuration* (in the case of a cycle graph, the sequence is infinite in both directions). A maximal sequence of adjacent nodes such that each node v has the same current symbol $c(v)$ is called a *block*. We sometimes identify a block with the subword consisting of the current symbols of the block nodes.

In the next two lemmas, we show how updating the current symbol locally in nodes results in global substitutions in the configuration.

▶ **Lemma 11.** *Assume that in the current configuration, each maximal subword of the form* $0+$ *or* $1+$ *is of length* ℓ *and each maximal subword of the form* $\_+$ *is of length 1. If the algorithm is executed for one phase and no node aborts, in the resulting configuration the lengths are* $4\ell + 3$ *and 1, respectively. More precisely, the execution of one phase always results in substitutions of the following kinds:*

$$\_0^\ell\_1^\ell\_1^\ell\_0^\ell\_1^\ell\_0^\ell\_0^\ell\_1^\ell\_ \mapsto \_0^{4\ell+3}\_1^{4\ell+3}\_, \tag{1}$$

$$\_1^\ell\_0^\ell\_0^\ell\_1^\ell\_0^\ell\_1^\ell\_1^\ell\_0^\ell\_ \mapsto \_1^{4\ell+3}\_0^{4\ell+3}\_. \tag{2}$$

Proof. Since no node aborts, each node v is able to find an unambiguous new value for $c(v)$. Consider an arbitrary node u. Suppose that the pattern $P_1 = \_0\_1\_1\_0\_1\_0\_0\_1\_$ matches $V(u)$ so that the ith symbol of the pattern is aligned with the $q(v)$th symbol of $V(u)$. Now it follows from Lemma 10 that P_1 is the compressed version of an actual neighbourhood N of u. Due to the assumption, for each j such that the jth symbol of P_1 is 0 or 1, in the neighbourhood there are exactly ℓ consecutive symbols 0 or 1, respectively, that are mapped to the jth symbol of P_1 by the compression mapping f.

Let us consider the case $i = 6$ as an example. Now $c(u) = 1$. As the buffers are gathered until each of them contain eight separators _, the next 5 blocks to the left from the block of u, as well as the next 11 blocks to the right, gather views that are compressed versions of a neighbourhood containing N. Hence P_1 matches also their views. For example, for all nodes w in the block two steps left from the block of u, P_1 matches $V(w)$ at position $i - 2 = 4$. It follows from the definition of the algorithm that the new value for $c(u)$, as well as for $c(w)$, where w is in the next 4 blocks to the left from u or 2 blocks to the right from u, is 0. The new value for the node in the 5th block left from u as well as 3rd block right from u is _. Thus, after the phase, node u will be part of a 0-block of length $\ell + 1 + \ell + 1 + \ell + 1 + \ell = 4\ell + 3$.

The cases for all other values of i, as for as the matching the reverse pattern P_2, are analogous. ◀

We call a word $\_x_1^i\_x_2^i\_\dots\_x_p^i\_$ a *padded Thue–Morse word of length* p if $x_1x_2\dots x_p$ is a prefix of the Thue–Morse sequence.

▶ **Lemma 12.** *Let W be a subword of a configuration C at the beginning of a phase. Let $k \geq 3$. If W is a (complement of a) padded Thue–Morse word of length 2^k and the algorithm is executed for one phase on C without aborting, the subword W is transformed to a (complement of a) padded Thue–Morse word of length 2^{k-2}.*

Proof. We use induction on k. If $k = 3$, we have $W = \_0^i\_1^i\_1^i\_0^i\_1^i\_0^i\_0^i\_1^i\_$ or $W = \_1^i\_0^i\_0^i\_1^i\_0^i\_1^i\_1^i\_0^i\_$ for some i. Then Lemma 11 implies that W is transformed to $\_0^j\_1^j\_$ or $\_1^j\_0^j\_$, respectively, where $j = 4i + 3$. Hence the claim holds for $k = 3$.

Suppose then that $k > 3$ and the claim holds for each subword that is a (complement of a) padded Thue–Morse word of length 2^{k-1}. Let W be a padded Thue–Morse word of length 2^k. Now the definition of the Thue–Morse sequence implies that we can write $W = W_1\_W_2$, where $W_1\_$ is a padded Thue–Morse word of length 2^{k-1} and $\_W_2$ is a complement of a padded Thue–Morse word of length 2^{k-1}. By the inductive hypothesis, $W_1\_$ is transformed to a padded Thue–Morse word of length 2^{k-3} and $\_W_2$ is transformed to a complement of a padded Thue–Morse word of length 2^{k-3}. Now the definition of the Thue–Morse sequence again implies that $W = W_1\_W_2$ is transformed to a padded Thue–Morse word of length 2^{k-2}. The case where W is a complement of a padded Thue–Morse word is completely analogous. ◀

Now we are ready to aggregate our previous lemmas to establish that algorithm $\mathcal{A}$ actually works correctly.

▶ **Lemma 13.** *In a yes-instance, each node eventually halts and outputs* yes*.*

Proof. Let (G, f) be a yes-instance of ThueMorse. By definition, algorithm $\mathcal{A}$ executes parts I and II successfully. Notice then that since the input word is valid, the configuration at the beginning of part III is actually a padded Thue–Morse word. If the input word is either _0_ or _0_1_1_0, the nodes move immediately to the yes state and halt. Otherwise, the input word is a padded Thue–Morse word of length at least 16, and by iterating Lemma 12 we obtain a shorter padded Thue–Morse word after every phase. Note that the new current symbol will be unambiguous for each node on each phase. Eventually the configuration will match with _0_1_1_0_, and the instance will get accepted. ◀

▶ **Lemma 14.** *In a no-instance, each node eventually halts and outputs* no*.*

Proof. Let (G, f) be a no-instance of ThueMorse. By assumption, G is finite and connected. If G contains a node of degree higher than 2, algorithm $\mathcal{A}$ aborts in part I. Hence we can assume that G has maximum degree at most two. It follows immediately from Lemma 11 that algorithm $\mathcal{A}$ halts on (G, f): since the size of blocks of 0's or 1's grows in each phase, we will eventually run out of nodes.

Graph G is either a path or a cycle graph. If $\mathcal{A}$ rejects in part I or part II, we are done. Hence, we can assume that the input f defines a consistent orientation and each 1-neighbourhood is of the correct form – that is, every second symbol on the input word is a separator _. Now it follows from Lemma 11 that the computation proceeds synchronously: in part III, each node starts a new phase in the same round.

Consider the case that G is a cycle graph. Due to Lemma 11, the number of blocks decreases after each phase. Eventually, if no node rejects, the execution reaches a configuration with $b \leq 4$ blocks of 0's and 1's. Then, each node sees the same sequence of b blocks repeating in its view. It follows that neither pattern P_1 or P_2 matches and the node rejects.

Assume then that G is a path graph. If there exists a yes-instance (G', f') with the same number of nodes as G has, we proceed as follows. Suppose for a contradiction that (G, f) gets accepted. Then there is a smallest i such that before the ith phase, the configurations C on (G, f) and C' on (G', f'), respectively, are different, but after the ith phase, they are identical, C''. It follows that C'' contains a subword of the form $\_0^i\_1^i\_$ for some i, such that C and C' differ on a position overlapping with the subword. But this is a contradiction, as it follows from Lemma 11 that C and C' cannot differ on such a position.

Finally, consider the case where (G, f) is a no-instance such that there does not exist a yes-instance with the same number of nodes. Suppose again for a contradiction that (G, f) gets accepted. Let (G', f') be the largest yes-instance no larger than (G, f) and let (G'', f'') be the yes-instance that is one step larger from (G', f'). Lemma 12 implies that the algorithm $\mathcal{A}$ needs exactly one phase more on (G'', f'') than on (G', f'). The number of phases on (G, f) equals either the number of phases on (G', f') or on (G'', f''). But either case is a contradiction, since the instances have a different amount of blocks in the beginning, and Lemma 11 implies that the size of blocks grows at a fixed rate. ◀

The last thing left to do is analysing the complexity of the algorithm. This is taken care of by the following lemmas.

▶ **Lemma 15.** *The space usage of algorithm $\mathcal{A}$ is in $O(1)$.*

Proof. Parts I and II of algorithm $\mathcal{A}$ clearly use only a constant number of states. Consider then part III. When gathering the buffers, consecutive blocks of symbols in the neighbourhood are represented by only one symbol, and only a constant amount of blocks are gathered (eight separator symbols _ in each buffer). Hence a constant number of states is enough to represent the contents of the buffers. Furthermore, the buffers get erased after each phase has completed. Thus, algorithm $\mathcal{A}$ can be implemented using a constant number of states, independent of the size of the input. ◀

▶ **Lemma 16.** *The running time of algorithm $\mathcal{A}$ is in $O(n)$.*

Proof. Executing each phase of part III of the algorithm can be done in $8i + 8 = 8(i + 1)$ rounds, where i is the size of the blocks at the start of the phase. Recall that after each phase, the size of the blocks grows from ℓ to $4\ell + 3$. Note also that $\frac{1}{2}\log n$ phases are enough to grow the block size so large that the algorithm has to either accept or reject. It follows that

$$8(1 + 1) + 8(7 + 1) + 8(31 + 1) + \cdots + 8(2^{2(\frac{1}{2}\log n)-1)}) \leq 8n$$

is an upper bound for the running time. ◀

On the other hand, $\Omega(n)$ rounds are clearly necessary to solve ThueMorse: the node at the end of the path has to receive information from the other end of the path to be able to verify that the instance is actually a yes-instance. This concludes the proof of Theorem 5.

5 Conclusions

We introduced space complexity as a new dimension in the classification of distributed graph problems. In particular, we identified a model of computation and a class of graphs, where the question on the existence of constant-space, non-constant-time algorithms is interesting and non-trivial. The answer turned out to be positive.

The result opens up the way to study constant space further – we can ask, for example, what other time complexities besides $\Theta(\log n)$ and $\Theta(n)$ can possibly be found and in which graph classes. Another open question of interest is the existence of natural graph problems that exhibit the constant-space non-constant-time characteristic.

Acknowledgements. We have discussed this work with numerous people, including at least Juho Hirvonen, Janne H. Korhonen, Antti Kuusisto, Joel Rybicki and Przemysław Uznański. Many thanks to all of them, as well as to others that we may have forgotten. We also thank the anonymous reviewers for their helpful feedback.

References

1 Yehuda Afek, Noga Alon, Ziv Bar-Joseph, Alejandro Cornejo, Bernhard Haeupler, and Fabian Kuhn. Beeping a maximal independent set. In *Proc. 25th International Symposium on Distributed Computing (DISC 2011)*, volume 6950 of *Lecture Notes in Computer Science*, pages 32–50. Springer, 2011. `doi:10.1007/978-3-642-24100-0_3`.

2 Dana Angluin. Local and global properties in networks of processors. In *Proc. 12th Annual ACM Symposium on Theory of Computing (STOC 1980)*, pages 82–93. ACM, 1980. `doi:10.1145/800141.804655`.

3 Alkida Balliu, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Dennis Olivetti, and Jukka Suomela. New classes of distributed time complexity, 2017. `arXiv:1711.01871`.

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 ACM Symposium on Theory of Computing (STOC 2016)*, pages 479–488. ACM, 2016. `doi:10.1145/2897518.2897570`. `arXiv:1511.00900`.

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. 36th Annual ACM Symposium on Principles of Distributed Computing (PODC 2017)*, pages 101–110. ACM, 2017. `doi:10.1145/3087801.3087833`. `arXiv:1702.05456`.

6 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. `doi:10.1109/FOCS.2016.72`. `arXiv:1602.08166`.

7 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)*, pages 156–167. IEEE, 2017. `doi:10.1109/FOCS.2017.23`. `arXiv:1704.06297`.

8 Lihi Cohen, Yuval Emek, Oren Louidor, and Jara Uitto. Exploring an infinite space with finite memory scouts. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*, pages 207–224. SIAM, 2017. `doi:10.1137/1.9781611974782.14`. `arXiv:1704.02380`.

9 Alejandro Cornejo and Fabian Kuhn. Deploying wireless networks with beeps. In *Proc. 24th International Symposium on Distributed Computing (DISC 2010)*, volume 6343 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2010. `doi:10.1007/978-3-642-15763-9_15`.

10 Yuval Emek, Tobias Langner, Jara Uitto, and Roger Wattenhofer. Solving the ANTS problem with asynchronous finite state machines. In *Proc. 41st International Colloquium on Automata, Languages, and Programming (ICALP 2014)*, pages 471–482. Springer, 2014. `doi:10.1007/978-3-662-43951-7_40`. `arXiv:1311.3062`.

11 Yuval Emek and Roger Wattenhofer. Stone age distributed computing. In *Proc. 32nd Annual ACM Symposium on Principles of Distributed Computing (PODC 2013)*, pages 137–146. ACM, 2013. doi:10.1145/2484239.2484244. arXiv:1202.1186.

12 Martin Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223(4):120–123, 1970.

13 Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. On the complexity of local distributed graph problems. In *Proc. 49th Annual ACM Symposium on Theory of Computing (STOC 2017)*, pages 784–797. ACM, 2017. doi:10.1145/3055399.3055471. arXiv:1611.02663.

14 Mika Göös, Juho Hirvonen, and Jukka Suomela. Lower bounds for local approximation. *Journal of the ACM*, 60(5):39:1–23, 2013. doi:10.1145/2528405. arXiv:1201.6675.

15 Lauri Hella, Matti Järvisalo, Antti Kuusisto, Juhana Laurinharju, Tuomo Lempiäinen, Kerkko Luosto, Jukka Suomela, and Jonni Virtema. Weak models of distributed computing, with connections to modal logic. *Distributed Computing*, 28(1):31–53, 2015. doi:10.1007/s00446-013-0202-3. arXiv:1205.2051.

16 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: lower and upper bounds. *Journal of the ACM*, 63(2):17:1–17:44, 2016. doi:10.1145/2742012. arXiv:1011.5470.

17 Antti Kuusisto. Infinite networks, halting and local algorithms. In *Proc. 5th International Symposium on Games, Automata, Logics and Formal Verification (GandALF 2014)*, volume 161 of *Electronic Proceedings in Theoretical Computer Science*, pages 147–160, 2014. doi:10.4204/EPTCS.161.14. arXiv:1408.5963.

18 Antti Kuusisto and Fabian Reiter. Emptiness problems for distributed automata. In *Proc. 8th International Symposium on Games, Automata, Logics and Formal Verification (GandALF 2017)*, volume 256 of *Electronic Proceedings in Theoretical Computer Science*, pages 210–222, 2017. doi:10.4204/EPTCS.256.15. arXiv:1705.02609.

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

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

21 Fabian Reiter. Distributed graph automata. In *Proc. 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2015)*, pages 192–201. IEEE, 2015. doi:10.1109/LICS.2015.27. arXiv:1404.6503.

22 Fabian Reiter. Asynchronous distributed automata: a characterization of the modal mu-fragment. In *Proc. 44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, pages 100:1–100:14. Schloss Dagstuhl – Leibniz Center for Informatics, 2017. doi:10.4230/LIPIcs.ICALP.2017.100. arXiv:1611.08554.

23 John von Neumann. *Theory of Self-Reproducing Automata.* University of Illinois Press, 1966.

24 Stephen Wolfram. *A New Kind of Science.* Wolfram Media, 2002.