Emptiness Problems for Distributed Automata

Antti Kuusisto
University of Bremen
Germany
antti.j.kuusisto@gmail.com

Fabian Reiter
IRIF, Université Paris Diderot
France
fabian.reiter@gmail.com

We investigate the decidability of the emptiness problem for three classes of distributed automata. These devices operate on finite directed graphs, acting as networks of identical finite-state machines that communicate in an infinite sequence of synchronous rounds. The problem is shown to be decidable in LOGSPACE for a class of forgetful automata, where the nodes see the messages received from their neighbors but cannot remember their own state. When restricted to the appropriate families of graphs, these forgetful automata are equivalent to classical finite word automata, but strictly more expressive than finite tree automata. On the other hand, we also show that the emptiness problem is undecidable in general. This already holds for two heavily restricted classes of distributed automata: those that reject immediately if they receive more than one message per round, and those whose state diagram must be acyclic except for self-loops.

1 Introduction

Recent years have seen increased interest in automata theoretic approaches to the study of distributed message-passing algorithms. Such algorithms are executed concurrently by all nodes of an arbitrary computer network in order to solve some graph problem related to the network structure. The weakest classes of these algorithms can be represented as deterministic finite-state machines, here referred to as *distributed automata*, which run as follows on a finite labeled directed graph: We place a copy of the same machine on every node of the graph and let the nodes communicate in an infinite sequence of synchronous rounds. In every round, each node computes its next local state as a function of its own current state and the set of current states of its incoming neighbors. (The states of the incoming neighbors represent incoming messages sent by the neighbors.) Acting as a semi-decider, the machine at a given node accepts precisely if it visits an accepting state at some point in time.

In a recently initiated research program, several classes of distributed algorithms have been given logical characterizations in the spirit of descriptive complexity theory [5], and conversely, some well-known logics have been provided with novel machine-oriented characterizations. First, in [3, 4], Hella et al. established the equivalence of *local* distributed automata and basic *modal logic*; in the context of distributed computing, the term "local" means that nodes stop changing their state after a constant number of rounds (see, e.g., [11]). The link with logic was further strengthened by Kuusisto in [6], where a logical characterization for unrestricted (nonlocal) automata was obtained in terms of a modal-logic-based variant of Datalog called *modal substitution calculus* (MSC). Then, in [9], Reiter extended local distributed automata with a global acceptance condition and the ability to alternate between nondeterministic and parallel computations, thereby providing an automata-theoretic characterization of *monadic second-order logic* (MSO) on arbitrary graphs. Similarly, the least fixpoint fragment of the modal μ-calculus has been characterized in [10] using an asynchronous subclass of nonlocal distributed automata. Furthermore, the descriptive complexity approach of [3, 4] and [6] found an application in [7], where tools from logic were used to show that universally halting distributed automata are necessarily local if we allow infinite networks into the picture.

P. Bouyer, A. Orlandini & P. San Pietro (Eds.): 8th Symposium on Games, Automata, Logics and Formal Verification (GandALF'17)
EPTCS 256, 2017, pp. 210–222, doi:10.4204/EPTCS.256.15

© A. Kuusisto & F. Reiter
This work is licensed under the Creative Commons Attribution License.

As the above equivalences are all effective, we can immediately settle the decidability question of the emptiness problem for local automata: it is decidable for the basic variant of [3, 4], but undecidable for the extension considered in [9]. This is because the (finite) satisfiability problem is PSPACE-complete for basic modal logic but undecidable for MSO. The problem is also decidable for the asynchronous class of [10], since (finite) satisfiability for the μ-calculus is EXPTIME-complete. However, the corresponding question for unrestricted automata was left open in [6]. In the present paper, we answer this question negatively for the general case and also consider it for three subclasses of distributed automata.

Our first variant, dubbed *forgetful* automata, is characterized by the fact that nodes can see their incoming neighbors' states but cannot remember their own state. Although this restriction might seem very artificial, it bears an intriguing connection to classical automata theory: forgetful distributed automata turn out to be equivalent to finite word automata (and hence MSO) when restricted to directed paths, but strictly more expressive than finite tree automata (and hence MSO) when restricted to ordered directed trees. As pointed out in [6, Prp. 8], the situation is different on arbitrary directed graphs, where distributed automata (and hence forgetful ones) are unable to recognize non-reachability properties that can be easily expressed in MSO. Hence, none of the two formalisms can simulate the other in general. However, while satisfiability for MSO is undecidable, we obtain a LOGSPACE algorithm that decides the emptiness problem for forgetful distributed automata.

The preceding decidability result begs the question of what happens if we drop the forgetfulness condition. Motivated by the equivalence of finite word automata and forgetful distributed automata on paths, we first investigate this question when restricted to directed paths. In sharp contrast to the forgetful case, we find that for arbitrary distributed automata, it is undecidable whether an automaton accepts on some directed path. Although our proof follows the standard approach of simulating a Turing machine, it has an unusual twist: we exchange the roles of space and time, in the sense that the space of the simulated Turing machine M is encoded into the time of the simulating distributed automaton A, and conversely, the time of M is encoded into the space of A. To lift this result to arbitrary graphs, we introduce the class of *monovisioned* distributed automata, where nodes enter a rejecting sink state as soon as they see more than one state in their incoming neighborhood. For every distributed automaton A, one can construct a monovisioned automaton A' that satisfies the emptiness property if and only if A does so on directed paths. Hence, the emptiness problem is undecidable for monovisioned automata, and thus also in general.

Our third and last class consists of those distributed automata whose state diagram does not contain any directed cycles, except for self-loops; we call them *quasi-acyclic*. The motivation for this particular class is threefold. First, quasi-acyclicity may be seen as a natural intermediate stage between local and unrestricted distributed automata, because local automata (for which the emptiness problem is decidable) can be characterized as those automata whose state diagram is acyclic as long as we ignore sink states (i.e., states that cannot be left once reached). Second, the Turing machine simulation mentioned above makes crucial use of directed cycles in the diagram of the simulating automaton, which suggests that cycles might be the source of undecidability. Third, the notion of quasi-acyclic state diagrams also plays a major role in [10], where it serves as an ingredient for the aforementioned subclass of asynchronous distributed automata (for which the emptiness problem is also decidable). However, contrary to what one might expect from these clues, we show that quasi-acyclicity alone is not sufficient to make the emptiness problem decidable, thereby giving an alternative proof of undecidability for the general case.

The remainder of this paper is organized as follows: We first introduce the formal definitions in Section 2 and establish the connections between forgetful distributed automata and classical word and tree automata in Section 3. Then, we show the positive decidability result for forgetful automata in Section 4. Finally, we establish the negative results for monovisioned automata in Section 5 and for quasi-acyclic automata in Section 6.

2 Preliminaries

We denote the set of non-negative integers by $\mathbb{N} = \{0, 1, 2, \dots\}$ and the power set of any set S by 2^S.

Let Σ be a finite set of symbols and r be a positive integer. A (finite) *Σ-labeled, r-relational directed graph*, abbreviated *digraph*, is a structure $G = (V, (E_k)_{1 \le k \le r}, \lambda)$, where V is a finite nonempty set of nodes, each $E_k \subseteq V \times V$ is a set of directed edges, and $\lambda : V \to \Sigma$ is a labeling that assigns a symbol of Σ to each node. Isomorphic digraphs are considered to be equal. If v is a node in V, we call the pair (G, v) a *pointed digraph* with *distinguished node* v. Furthermore, if uv is an edge in E_k, then u is called an *incoming k-neighbor* of v, or simply an *incoming neighbor*.

A *directed rooted tree*, or *ditree*, is a digraph $G = (V, (E_k)_{1 \le k \le r}, \lambda)$ that has a distinct node v_ε, called the *root*, such that from each node v in V, there is exactly one way to reach v_ε by following the directed edges in $\bigcup_{1 \le k \le r} E_k$, where $E_i \cap E_j = \emptyset$ for $i \neq j$. A *pointed ditree* is a pointed digraph (G, v_ε) that is composed of a ditree and its root. Moreover, an r-relational ditree is called *ordered* if for $1 \le k \le r$, every node has at most one incoming k-neighbor and every node that has an incoming $(k+1)$-neighbor also has an incoming k-neighbor. As a special case, an ordered 1-relational ditree is referred to as a *directed path*, or *dipath*.

We now give a general definition of distributed automata that subsumes all the variants considered in this paper. Simply put, a distributed automaton is a deterministic finite-state machine that reads sets of states instead of the usual alphabetic symbols. To run such an automaton on a digraph, we place a copy of the same machine on every node of the digraph and let the nodes communicate in an infinite sequence of synchronous rounds. In every round, each node computes its next local state as a function of its own current state and the set of current states of its incoming neighbors. In order to draw the comparison with classical word and tree automata in Section 3, we let our distributed automata operate on labeled, multi-relational digraphs. Furthermore, we let the nodes of those digraphs read their own label in each communication round, as this will facilitate the definition of forgetful automata. Whenever possible, the rather cumbersome notation will later be simplified.

Definition 1 (Distributed Automaton)**.** A *distributed automaton* over Σ-labeled, r-relational digraphs is a tuple $A = (Q, q_0, (\delta_a)_{a \in \Sigma}, F)$, where Q is a finite nonempty set of states, $q_0 \in Q$ is an initial state, $\delta_a \colon Q \times (2^Q)^r \to Q$ is a (local) transition function associated with label $a \in \Sigma$, and $F \subseteq Q$ is a set of accepting states.

Let $G = (V, (E_k)_{1 \le k \le r}, \lambda)$ be a Σ-labeled, r-relational digraph. The *run* of A on G is an infinite sequence $\rho = (\rho_0, \rho_1, \rho_2, \dots)$ of maps $\rho_t : V \to Q$, called *configurations*, which are defined inductively as follows, for $t \in \mathbb{N}$ and $v \in V$:

$$\rho_0(v) = q_0 \qquad \text{and} \qquad \rho_{t+1}(v) = \delta_{\lambda(v)}\Big(\rho_t(v),\, \big(\{\rho_t(u) \mid uv \in E_k\}\big)_{1 \le k \le r}\Big).$$

For $v \in V$, the automaton A *accepts* the pointed digraph (G, v) if v visits an accepting state at some point in the run ρ of A on G, i.e., if there exists $t \in \mathbb{N}$ such that $\rho_t(v) \in F$. The *language* of A (or language recognized by A) is the set of all pointed digraphs that A accepts.

A distributed automaton is called *forgetful* if in each round, the nodes can see their neighbors' states but cannot remember their own state. Formally, for $A = (Q, q_0, (\delta_a)_{a \in \Sigma}, F)$, being forgetful means that $\delta_a(q, \vec{S}) = \delta_a(q', \vec{S})$ for all $a \in \Sigma$, $q, q' \in Q$ and $\vec{S} \in (2^Q)^r$. Therefore, we can represent the transition functions of such an automaton as $\delta_a \colon (2^Q)^r \to Q$.

On the other hand, when we consider automata that are not forgetful, we will simplify them to have a single transition function. Instead of letting the nodes read their own label a and choose the appropriate function δ_a in each round, we can force them to store the label in their local state and combine all the

transition functions into a single one. Notation can be further lightened by limiting ourselves to 1-relational digraphs. Hence, we shall sometimes regard a distributed automaton as a tuple $A = (Q, \delta_0, \delta, F)$, where $\delta_0 \colon \Sigma \to Q$ is an initialization function, $\delta \colon Q \times 2^Q \to Q$ is a transition function, and Q and F are as before. The semantics is the obvious one: each node v is initialized to $\delta_0(\lambda(v))$, computes its next state by evaluating δ on its current state and the set of states of its incoming neighbors, and accepts if at some point in time it visits a state in F.

The central concern of this paper is the (general) *emptiness problem* for several classes of distributed automata. Given an automaton A, the problem is to decide effectively whether the language of A is nonempty, i.e., whether there is a pointed digraph (G, v) that is accepted by A. Similarly, the *dipath-emptiness problem* is to decide if A accepts some pointed dipath.

3 Comparison with classical automata

The purpose of this section is to motivate our interest in forgetful distributed automata by establishing their connection with classical word and tree automata.

Proposition 2. *When restricted to the class of pointed dipaths, forgetful distributed automata are equivalent to finite word automata (and thus to monadic second-order logic).*

Proof. Let us denote a (deterministic) finite word automaton over some finite alphabet Σ by a tuple $B = (P, p_0, \tau, H)$, where P is the set of states, p_0 is the initial state, $\tau \colon P \times \Sigma \to P$ is the transition function, and H is the set of accepting states.

Given such a word automaton B, we construct a forgetful distributed automaton $A = (Q, q_0, (\delta_a)_{a \in \Sigma}, F)$ that simulates B on Σ-labeled dipaths. For this, it suffices to set $Q = P \cup \{\bot\}$, $q_0 = \bot$, $F = H$, and

$$\delta_a(S) = \begin{cases} \tau(p_0, a) & \text{if } S = \emptyset, \\ \tau(p, a) & \text{if } S = \{p\} \text{ for some } p \in P, \\ \bot & \text{otherwise.} \end{cases}$$

When A is run on a dipath, each node v starts in a waiting phase, represented by $\bot$, and remains idle until its predecessor has computed the state p that B would have reached just before reading the local symbol a of v. (If there is no predecessor, p is set to p_0.) Then, v switches to the state $\tau(p, a)$ and stays there forever. Consequently, the distinguished last node of the dipath will end up in the state reached by B at the end of the word, and it accepts if and only if B does.

For the converse direction, we convert a given forgetful distributed automaton $A = (Q, q_0, (\delta_a)_{a \in \Sigma}, F)$ into the word automaton $B = (P, p_0, \tau, H)$ with components $P = 2^Q$, $p_0 = \emptyset$, $H = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$, and

$$\tau(p, a) = \{q_0\} \cup \begin{cases} \{\delta_a(\emptyset)\} & \text{if } p = p_0, \\ \{\delta_a(\{q\}) \mid q \in p\} & \text{otherwise.} \end{cases}$$

On any Σ-labeled dipath G, our construction guarantees that the set of states visited by A at the i-th node is equal to the state that B reaches just after processing the i-th symbol of the word associated with G. We can easily verify this by induction on i: At the first node, which is labeled with a_1, automaton A starts in state q_0 and then remains forever in state $\delta_{a_1}(\emptyset)$. Node number $i+1$ also starts in q_0, and transitions to $\delta_{a_{i+1}}(\{q_t^i\})$ at time $t+1$, where a_{i+1} is the node's own label and q_t^i is the state of its predecessor at time t. In agreement with this behavior, we know by the induction hypothesis and the definition of τ that the state

of B after reading a_{i+1} is precisely $\{q_0\} \cup \{\delta_{a_{i+1}}(\{q_t^i\}) \mid t \in \mathbb{N}\}$. As a result, the final state reached by B will be accepting if and only if A visits some accepting state at the last node. □

A (deterministic, bottom-up) finite tree automaton over Σ-labeled, r-relational ordered ditrees can be defined as a tuple $B = (P, (\tau_k)_{0 \le k \le r}, H)$, where P is a finite nonempty set of states, $\tau_k \colon P^k \times \Sigma \to P$ is a transition function of arity k, and $H \subseteq P$ is a set of accepting states. Such an automaton assigns a state of P to each node of a given pointed ditree, starting from the leaves and working its way up to the root. If node v is labeled with symbol a and its k children have been assigned the states $p_1, \ldots, p_k$ (following the numbering order of the k first edge relations), then v is assigned the state $\tau_k(p_1, \ldots, p_k, a)$. Note that leaves are covered by the special case $k = 0$. Based on this, the pointed ditree is accepted if and only if the state at the root belongs to H. For a more detailed presentation see, e.g., [8, § 3.3].

Proposition 3. *When restricted to the class of pointed ordered ditrees, forgetful distributed automata are strictly more expressive than finite tree automata (and thus than monadic second-order logic).*

Proof. To convert a tree automaton $B = (P, (\tau_k)_{0 \le k \le r}, H)$ into a forgetful distributed automaton $A = (Q, q_0, (\delta_a)_{a \in \Sigma}, F)$ that is equivalent to B over Σ-labeled, r-relational ordered ditrees, we use a simple generalization of the construction in the proof of Proposition 2: $Q = P \cup \{\bot\}$, $q_0 = \bot$, $F = H$, and

$$\delta_a(\vec{S}) = \begin{cases} \tau_k(p_1, \ldots, p_k, a) & \text{if } \vec{S} = \big(\{p_1\}, \ldots, \{p_k\}, \emptyset, \ldots, \emptyset\big) \text{ for some } p_1, \ldots, p_k \in P, \\ \bot & \text{otherwise.} \end{cases}$$

In contrast, a conversion in the other direction is not always possible, as can be seen from the following example on binary ditrees. Consider the forgetful distributed automaton $A' = (\{\bot, \top, \star\}, \bot, \delta, \{\star\})$, with

$$\delta(S_1, S_2) = \begin{cases} \bot & \text{if } S_1 = S_2 = \{\bot\} \\ \top & \text{if } S_1, S_2 \in \{\emptyset, \{\top\}\} \\ \star & \text{otherwise.} \end{cases}$$

When run on an unlabeled, 2-relational ordered ditree, A' accepts at the root precisely if the ditree is *not* perfectly balanced, i.e., if there exists a node whose left and right subtrees have different heights. To achieve this, each node starts in the waiting state $\bot$, where it remains as long as it has two children and those children are also in $\bot$. If the ditree is perfectly balanced, then all the leaves switch permanently from $\bot$ to $\top$ in the first round, their parents do so in the second round, their parents' parents in the third round, and so forth, until the signal reaches the root. Therefore, the root will transition directly from $\bot$ to $\top$, never visiting state $\star$, and hence the pointed ditree is rejected. On the other hand, if the ditree is not perfectly balanced, then there must be some lowermost internal node v that does not have two subtrees of the same height (in particular, it might have only one child). Since its subtrees are perfectly balanced, they behave as in the preceding case. At some point in time, only one of v's children will be in state $\bot$, at which point v will switch to state $\star$. This triggers an upward-propagating chain reaction, eventually causing the root to also visit $\star$, and thus to accept. Note that $\star$ is just an intermediate state; regardless of whether or not the ditree is perfectly balanced, every node will ultimately end up in $\top$.

To prove that A' is not equivalent to any tree automaton, one can simply invoke the pumping lemma for regular tree languages to show that the complement language of A' is not recognizable by any tree automaton. The claim then follows from the fact that regular tree languages are closed under complementation. □

4 Exploiting forgetfulness

We now give an algorithm deciding the emptiness problem for forgetful distributed automata (on arbitrary digraphs). Its space complexity is linear in the number of states of the given automaton. However, as an uncompressed binary encoding of a distributed automaton requires space exponential in the number of states, this results in LOGSPACE complexity. Obviously, the statement might not hold anymore if the automaton were instead represented by a more compact device, such as a logical formula.

Theorem 4. *The emptiness problem for forgetful distributed automata is decidable in* LOGSPACE.

Proof. Let $A = (Q, q_0, (\delta_a)_{a\in\Sigma}, F)$ be some forgetful distributed automaton over Σ-labeled, r-relational digraphs. Consider the infinite sequence of sets of states $S_0, S_1, S_2 \cdots$ such that S_t contains precisely those states that can be visited by A at some node in some digraph at time t. That is, $q \in S_t$ if and only if there exists a pointed digraph (G, v) such that $\rho_t(v) = q$, where ρ is the run of A on G. From this point of view, the language of A is nonempty precisely if there is some $t \in \mathbb{N}$ for which $S_t \cap F \neq \emptyset$.

By definition, we have $S_0 = \{q_0\}$. Furthermore, exploiting the fact that A is forgetful, we can specify a simple function $\Delta\colon 2^Q \to 2^Q$ such that $S_{t+1} = \Delta(S_t)$:

$$\Delta(S) = \left\{ \delta_a(\vec{T}) \;\middle|\; a \in \Sigma \text{ and } \vec{T} \in (2^S)^r \right\}$$

Obviously, $S_{t+1} \subseteq \Delta(S_t)$. To see that $S_{t+1} \supseteq \Delta(S_t)$, assume we are given a pointed digraph (G_q, v_q) for each state $q \in S_t$ such that v_q visits q at time t in the run of A on G_q. (Such a pointed digraph must exist by the definition of S_t.) Now, for any $a \in \Sigma$ and $\vec{T} = (T_1, \ldots, T_r) \in (2^{S_t})^r$, we construct a new digraph G as follows: Starting with a single a-labeled node v, we add a (disjoint) copy of G_q for each state q that occurs in some set T_k. Then, we add a k-edge from v_q to v if and only if $q \in T_k$. Each node v_q behaves the same way in G as in G_q because v has no influence on its incoming neighbors. Since A is forgetful, the state of v at time $t+1$ depends solely on its own label and its incoming neighbor's states at time t. Consequently, v visits the state $\delta_a(\vec{T})$ at time $t+1$, and thus $\delta_a(\vec{T}) \in S_{t+1}$.

Now, we know that the sequence $S_0, S_1, S_2 \cdots$ must be eventually periodic because its generator function Δ maps the finite set 2^Q to itself. Hence, it suffices to consider the prefix of length $|2^Q|$ in order to determine whether $S_t \cap F \neq \emptyset$ for some $t \in \mathbb{N}$. This leads to the following simple algorithm, which decides the emptiness problem for forgetful automata.

EMPTY(A): $S \leftarrow \{q_0\}$
repeat at most $|2^Q|$ times :
$S \leftarrow \Delta(S)$
if $S \cap F \neq \emptyset$: **return** true
return false

It remains to analyze the space complexity of this algorithm. For that, we assume that the binary encoding of A given to the algorithm contains a lookup table for each transition function δ_a and a bit array representing F, which amounts to an asymptotic size of $\Theta\big(|\Sigma| \cdot |2^Q|^r \cdot \log|Q|\big)$ input bits. To implement the procedure EMPTY, we need $|Q|$ bits of working memory to represent the set S and another $|Q|$ bits for the loop counter. Furthermore, we can compute $\Delta(S)$ for any given set $S \subseteq Q$ by simply iterating over all $a \in \Sigma$ and $\vec{T} \in (2^Q)^r$, and adding $\delta_a(\vec{T})$ to the returned set if all components of $\vec{T}$ are subsets of S. This requires $\log|\Sigma| + |Q| \cdot r$ additional bits to keep track of the iteration progress, $\Theta\big(\log|\Sigma| + |Q| \cdot r + \log\log|Q|\big)$ bits to store pointers into the lookup tables, and $|Q|$ bits to store the intermediate result. In total, the algorithm uses $\Theta\big(\log|\Sigma| + |Q| \cdot r\big)$ bits of working memory, which is logarithmic in the size of the input. □

5 Exchanging space and time

In this section, we first show the undecidability of the dipath-emptiness problem for arbitrary distributed automata, and then lift that result to the general emptiness problem.

Theorem 5. *The dipath-emptiness problem for distributed automata is undecidable.*

Proof sketch. We proceed by reduction from the halting problem for Turing machines. For our purposes, a Turing machine operates deterministically with one head on a single tape, which is one-way infinite to the right and initially empty. The problem consists of determining whether the machine will eventually reach a designated halting state. We show a way of encoding the computation of a Turing machine M into the run of a distributed automaton A over unlabeled digraphs, such that the language of A contains a pointed dipath if and only if M reaches its halting state.

Note that since dipaths are oriented, the communication between their nodes is only one-way. Hence, we cannot simply represent (a section of) the Turing tape as a dipath. Instead, the key idea of our simulation is to exchange the roles of space and time, in the sense that the space of M is encoded into the time of A, and the time of M into the space of A. Assuming the language of A contains a dipath, we will think of that dipath as representing the timeline of M, such that each node corresponds to a single point in time in the computation of M. Roughly speaking, when running A, the node v_t corresponding to time t will "traverse" the configuration C_t of M at time t. Here, "traversing" means that the sequence of states of A visited by v_t is an encoding of C_t read from left to right, supplemented with some additional bookkeeping information.

The first element of the dipath, node v_0, starts by visiting a state of A representing an empty cell that is currently read by M in its initial state. Then it transitions to another state that simply represents an empty cell, and remains in such a state forever after. Thus v_0 does indeed "traverse" C_0. We will show that it is also possible for any other node v_t to "traverse" its corresponding configuration C_t, based on the information it receives from v_{t-1}. In order for this to work, we shall give v_{t-1} a head start of two cells, so that v_t can compute the content of cell i in C_t based on the contents of cells $i-1$, i and $i+1$ in C_{t-1}.

Node v_t enters an accepting state of A precisely if it "sees" the halting state of M during its "traversal" of C_t. Hence, A accepts the pointed dipath of length t if and only if M reaches its halting state at time t.

We now describe the inner workings of A in a semi-formal way. In parallel, the reader might want to have a look at Figure 1, which illustrates the construction by means of an example. Let M be represented by the tuple $(P,\Gamma,p_0,\square,\tau,p_\mathrm{h})$, where P is the set of states, Γ is the tape alphabet, p_0 is the initial state, $\square$ is the blank symbol, $\tau\colon (P\setminus\{p_\mathrm{h}\})\times\Gamma\to P\times\Gamma\times\{\mathrm{L},\mathrm{R}\}$ is the transition function, and p_h is the halting state. From this, we construct A as (Q,q_0,δ,F), with the state set $Q=(\{\bot\}\cup(P\times\Gamma)\cup\Gamma)^3$, the initial state $q_0=(\bot,\bot,\bot)$, the transition function δ specified informally below, and the accepting set F that contains precisely those states that have p_h in their third component. In keeping with the intuition that each node of the dipath "traverses" a configuration of M, the third component of its state indicates the content of the "currently visited" cell i. The two preceding components keep track of the recent history, i.e., the second component always holds the content of the previous cell $i-1$, and the first component that of $i-2$. In the following explanation, we concentrate on updating the third component, tacitly assuming that the other two are kept up to date. The special symbol $\bot$ indicates that no cell has been "visited", and we say that a node is in the waiting phase while its third component is $\bot$.

In the first round, v_0 sees that it does not have any incoming neighbor, and thus exits the waiting phase by setting its third component to $(p_0,\square)$, and after that, it sets it to $\square$ for the remainder of the run. Every other node v_t remains in the waiting phase as long as its incoming neighbor's second component is $\bot$. This ensures a delay of two cells with respect to v_{t-1}. Once v_t becomes active, given the *current* state

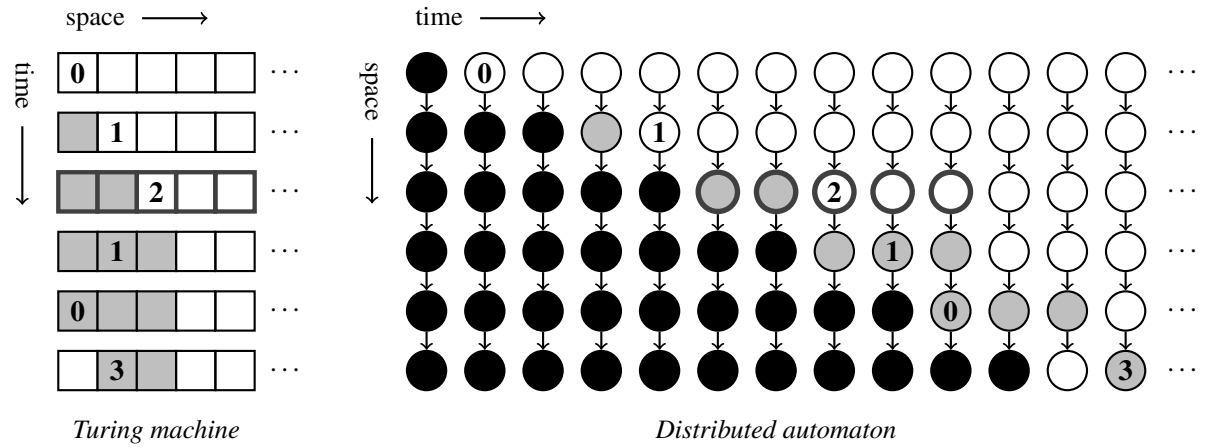

Figure 1. Exchanging space and time to prove Theorem 5. The left-hand side depicts the computation of a Turing machine with state set $\{\mathbf{0},\mathbf{1},\mathbf{2},\mathbf{3}\}$ and tape alphabet $\{\square,\blacksquare\}$. On the right-hand side, this machine is simulated by a distributed automaton run on a dipath. Waiting nodes are represented in black, whereas active nodes display the content of the "currently visited" cell of the Turing machine (i.e., only the third component of the states is shown).

(c_1,c_2,c_3) of v_{t-1}, it computes the third component d_3 of its own *next* state (d_1,d_2,d_3) as follows: If none of the components c_1, c_2, c_3 "contain the head of M", i.e., if none of them lie in $P\times\Gamma$, then it simply sets d_3 to be equal to c_2. Otherwise, a computation step of M is simulated in the natural way. For instance, if c_3 is of the form (p,γ), and $\tau(p,\gamma)=(p',\gamma',\mathrm{L})$, then d_3 is set to (p',c_2). This corresponds to the case where, at time $t-1$, the head of M is located to the right of v_t's next "position" and moves to the left. As another example, if c_2 is of the form (p,γ), and $\tau(p,\gamma)=(p',\gamma',\mathrm{R})$, then d_3 is set to γ'. The remaining cases are handled analogously.

Note that, thanks to the two-cell delay between adjacent nodes, the head of M always "moves forward" in the time of A, although it may move in both directions with respect to the space of M (see Figure 1). □

To infer from Theorem 5 that the general emptiness problem for distributed automata is also undecidable, we now introduce the notion of *monovisioned* automata, which have the property that nodes "expect" to see no more than one state in their incoming neighborhood at any given time. More precisely, a distributed automaton $A=(Q,\delta_0,\delta,F)$ is monovisioned if it has a rejecting sink state $q_{\mathrm{rej}}\in Q\setminus F$, such that $\delta(q,S)=q_{\mathrm{rej}}$ whenever $|S|>1$ or $q_{\mathrm{rej}}\in S$ or $q=q_{\mathrm{rej}}$, for all $q\in Q$ and $S\subseteq Q$. Obviously, for every distributed automaton, we can construct a monovisioned automaton that has the same acceptance behavior on dipaths. Furthermore, as shown by means of the next two lemmas, the emptiness problem for monovisioned automata is equivalent to its restriction to dipaths. All put together, we get the desired reduction from the dipath-emptiness problem to the general emptiness problem.

Lemma 6. *The language of a distributed automaton is nonempty if and only if it contains a pointed ditree.*

Proof sketch. We slightly adapt the notion of *tree-unraveling*, which is a standard tool in modal logic (see, e.g., [2, Def. 4.51] or [1, § 3.2]). Consider any distributed automaton A. Assume that A accepts some pointed digraph (G,v), and let $t\in\mathbb{N}$ be the first point in time at which v visits an accepting state. Based on that, we can easily construct a pointed ditree (G',v') that is also accepted by A. First of all, the root v' of G' is chosen to be a copy of v. On the next level of the ditree, the incoming neighbors of v' are chosen to be fresh copies $u'_1,\dots,u'_n$ of v's incoming neighbors $u_1,\dots,u_n$. Similarly, the incoming neighbors of $u'_1,\dots,u'_n$ are fresh copies of the incoming neighbors of $u_1,\dots,u_n$. If u_i and u_j have incoming neighbors

in common, we create distinct copies of those neighbors for u'_i and u'_j. This process is iterated until we obtain a ditree of height t. It is easy to check that v and v' visit the same sequence of states $q_0, q_1, \ldots, q_t$ during the first t communication rounds. □

Lemma 7. *The language of a monovisioned distributed automaton is nonempty if and only if it contains a pointed dipath.*

Proof sketch. Consider any monovisioned distributed automaton A whose language is nonempty. By Lemma 6, A accepts some pointed ditree (G, v). Let $t \in \mathbb{N}$ be the first point in time at which v visits an accepting state. Now, it is easy to prove by induction that for all $i \in \{0, \ldots, t\}$, sibling nodes at depth i traverse the same sequence of states $q_0, q_1, \ldots, q_{t-i}$ between times 0 and $t - i$, and this sequence does not contain the rejecting state q_{rej}. Thus, A also accepts any dipath from some node at depth t to the root. □

6 Timing a firework show

We now show that the emptiness problem is undecidable even for quasi-acyclic automata. This also provides an alternative, but more involved undecidability proof for the general case.

A distributed automaton $A = (Q, \delta_0, \delta, F)$ is said to be *quasi-acyclic* if its state diagram does not contain any directed cycles, except for self-loops. More formally, this means that for every sequence $q_1, q_2, \ldots, q_n$ of states in Q such that $q_1 = q_n$ and $\delta(q_i, S_i) = q_{i+1}$ for some $S_i \subseteq Q$, it must hold that all states of the sequence are the same. Notice that our proof of Theorem 5 does not go through if we consider only quasi-acyclic automata.

It is straightforward to see that quasi-acyclicity is preserved under a standard product construction, similar to the one employed for finite automata on words. Hence, we have the following closure property, which will be used in the subsequent undecidability proof.

Lemma 8. *The class of languages recognizable by quasi-acyclic distributed automata is closed under union and intersection.*

Theorem 9. *The emptiness problem for quasi-acyclic distributed automata is undecidable.*

Proof sketch. We show this by reduction from Post's correspondence problem (PCP). An instance P of PCP consists of a collection of pairs of nonempty finite words $(x_i, y_i)_{i \in I}$ over the alphabet $\{0, 1\}$, indexed by some finite set of integers I. It is convenient to view each pair (x_i, y_i) as a domino tile labeled with x_i on the upper half and y_i on the lower half. The problem is to decide if there exists a nonempty sequence $S = (i_1, \ldots, i_n)$ of indices in I, such that the concatenations $x_S = x_{i_1} \cdots x_{i_n}$ and $y_S = y_{i_1} \cdots y_{i_n}$ are equal. We construct a quasi-acyclic automaton A whose language is nonempty if and only if P has such a solution S.

Metaphorically speaking, our construction can be thought of as a perfectly timed "firework show", whose only "spectator" will see a putative solution $S = (i_1, \ldots, i_n)$, and be able to check whether it is indeed a valid solution of P. Our "spectator" is the distinguished node v_ε of the pointed digraph on which A is run. We assume that v_ε has n incoming neighbors, one for each element of S. Let v_k denote the neighbor corresponding to i_k, for $1 \leq k \leq n$. Similarly to our proof of Theorem 5, we use the time of A to represent the spatial dimension of the words x_S and y_S. On an intuitive level, v_ε will "witness" simultaneous left-to-right traversals of x_S and y_S, advancing by one bit per time step, and it will check that the two words match. It is the task of each node v_k to send to v_ε the required bits of the subwords x_{i_k} and y_{i_k} at the appropriate times. In keeping with the metaphor of fireworks, the correct timing can be achieved by attaching to v_k a carefully chosen "fuse", which is "lit" at time 0. Two separate "fire" signals

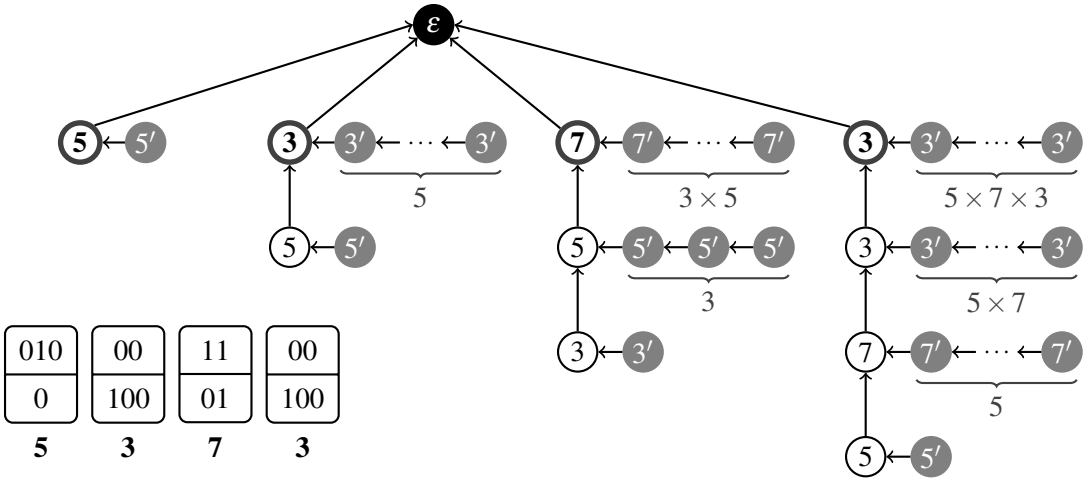

Figure 2. Timing a "firework show" to prove Theorem 9. The domino tiles on the bottom-left visualize the solution $(5,3,7,3)$ for the instance $\{3\mapsto(00,100), 5\mapsto(010,0), 7\mapsto(11,01)\}$ of PCP. This solution is encoded into the labeled ditree above, with node types ε, 3, 5, 7, $3'$, $5'$, $7'$. Each domino is represented by a bold-highlighted white node of the appropriate type. The "fuse" of such a bold node consists of the chain of white nodes below it, which lists the indices of the preceding dominos in an arbitrary order. Each white node also has a gray "side fuse" whose length is equal to the product of the white types occurring below that node. The "firework show" observed at the root will feature two simultaneous bitstreams, which both represent the sequence 010001100.

will travel at different speeds along this (admittedly sophisticated) "fuse", and once they reach v_k, they trigger the "firing" of x_{i_k} and y_{i_k}, respectively.

We now go into more details. Using the labeling of the input graph, the automaton A distinguishes between $2|I|+1$ different types of nodes: two types i and i' for each index $i \in I$, and one additional type ε to identify the "spectator". Motivated by Lemma 6, we suppose that the input graph is a pointed ditree, with a very specific shape that encodes a putative solution $S = (i_1, \dots, i_n)$. An example illustrating the following description of such a ditree-encoding is given in Figure 2. Although A is not able to enforce all aspects of this particular shape, we will make sure that it accepts such a structure if its language is nonempty. The root (and distinguished node) v_ε is the only node of type ε. Its children $v_1, \dots, v_n$ are of types $i_1, \dots, i_n$, respectively. The "fuse" attached to each child v_k is a chain of $k-1$ nodes that represents the multiset of indices occurring in the $(k-1)$-prefix of S. More precisely, there is an induced dipath $v_{k,1} \to \cdots v_{k,k-1} \to v_k$, such that the multiset of types of the nodes $v_{k,1}, \dots, v_{k,k-1}$ is equal to the multiset of indices occurring in $(i_1, \dots, i_{k-1})$. We do not impose any particular order on those nodes. Finally, each node of type $i \in I$ also has an incoming chain of nodes of type i' (depicted in gray in Figure 2), whose length corresponds exactly to the product of the types occurring on the part of the "fuse" below that node. That is, if we define the alias $v_{k,k} := v_k$, then for every node $v_{k,j}$ of type $i \in I$, there is an induced dipath $v_{k,j,1} \to \cdots v_{k,j,\ell} \to v_{k,j}$, where all the nodes $v_{k,j,1}, \dots, v_{k,j,\ell}$ are of type i', and the number ℓ is equal to the product of the types of the nodes $v_{k,1}, \dots, v_{k,j-1}$ (which is 1 if $j = 1$). We shall refer to such a chain $v_{k,j,1}, \dots, v_{k,j,\ell}$ as a "side fuse".

The automaton A has to perform two tasks simultaneously: First, assuming it is run on a ditree-encoding of a sequence S, exactly as specified above, it must verify that S is a valid solution, i.e., that the words x_S and y_S match. Second, it must ensure that the input graph is indeed sufficiently similar to such a ditree-encoding. In particular, it has to check that the "fuses" used for the first task are consistent with each other. Since, by Lemma 8, quasi-acyclic distributed automata are closed under intersection, we can

consider the two tasks separately, and implement them using two independent automata A_1 and A_2. In the following, we describe both devices in a rather informal manner. The important aspect to note is that they can be easily formalized using quasi-acyclic state diagrams.

We start with A_1, which verifies the solution S. It takes into account only nodes with types in $I \cup \{\varepsilon\}$ (thus ignoring the gray nodes in Figure 2). At nodes of type $i \in I$, the states of A_1 have two components, associated with the upper and lower halves of the domino (x_i, y_i). If a node of type i sees that it does not have any incoming neighbor, then the upper and lower components of its state immediately start traversing sequences of substates representing the bits of x_i and y_i, respectively. Since those substates must keep track of the respective positions within x_i and y_i, none of them can be visited twice. After that, both components loop forever on a special substate $\top$, which indicates the end of transmission. The other nodes of type i keep each of their two components in a waiting status, indicated by another substate $\bot$, until the corresponding component of their incoming neighbor reaches its last substate before $\top$. This constitutes the aforementioned "fire" signal. Thereupon, they start traversing the same sequences of substates as in the previous case. Note that both components are updated independently of each other, hence there can be an arbitrary time lag between the "traversals" of x_i and y_i. Now, assuming the "fuse" of each node v_k really encodes the multiset of indices occurring in $(i_1, \dots, i_{k-1})$, the delay accumulated along that "fuse" will be such that v_k starts "traversing" x_{i_k} and y_{i_k} at the points in time corresponding to their respective starting positions within x_S and y_S. That is, for x_{i_k} it starts at time $|x_{i_1} \cdots x_{i_{k-1}}| + 1$, and for y_{i_k} at time $|y_{i_1} \cdots y_{i_{k-1}}| + 1$. Consequently, in each round $t \leq \min\{|x_S|, |y_S|\}$, the root v_ε receives the t-th bits of x_S and y_S. At most two distinct children send bits at the same time, while the others remain in some state $q \in \{\bot, \top\}^2$. With this, the behavior of A_1 at v_ε is straightforward: It enters its only accepting state precisely if all of its children have reached the state $(\top, \top)$ and it has never seen any mismatch between the upper and lower bits.

We now turn to A_2, whose job is to verify that the "fuses" used by A_1 are reliable. Just like A_1, it works under the assumption that the input graph is a ditree as specified previously, but with significantly reduced guarantees: The root could now have an arbitrary number of children, the "fuses" and "side fuses" could be of arbitrary lengths, and each "fuse" could represent an arbitrary multiset of indices in I. Again using an approach reminiscent of fireworks, we devise a protocol in which each child v will send two distinct signals to the root v_ε. The first signal $\uparrow_1$ indicates that the current time t is equal to the product of the types of all the nodes on v's "fuse". Similarly, the second signal $\uparrow_2$ indicates that the current time is equal to that same product multiplied by v's own type. To achieve this, we make use of the "side fuses", along which two additional signals $\leftarrow_1$ and $\leftarrow_2$ are propagated. For each node of type $i \in I$, the nodes of type i' on the corresponding "side fuse" operate in a way such that $\leftarrow_1$ advances by one node per time step, whereas $\leftarrow_2$ is delayed by i time units at every node. Hence, $\leftarrow_1$ travels i times faster than $\leftarrow_2$. Building on that, each node v of type i (not necessarily a child of the root) sends $\uparrow_1$ to its parent, either at time 1, if it does not have any predecessor on the "fuse", or one time unit before receiving $\uparrow_2$ from its predecessor. The latter is possible, because the predecessor also sends a pre-signal $\uparrow_2^{\mathrm{pre}}$ before sending $\uparrow_2$. Then, v checks that signal $\leftarrow_1$ from its "side fuse" arrives exactly at the same time as $\uparrow_2$ from its predecessor, or at time 1 if there is no predecessor. Otherwise, it immediately enters a rejecting state. This will guarantee, by induction, that the length of the "side fuse" is equal to the product of the types on the "fuse" below. Finally, two rounds prior to receiving $\leftarrow_2$, while that signal is still being delayed by the last node on the "side fuse", v first sends the pre-signal $\uparrow_2^{\mathrm{pre}}$, and then the signal $\uparrow_2$ in the following round. For this to work, we assume that each node on the "side fuse" waits for at least two rounds between receiving $\leftarrow_2$ from its predecessor and forwarding the signal to its successor, i.e., all indices in I must be strictly greater than 2. Due to the delay accumulated by $\leftarrow_2$ along the "side fuse", the time at which $\uparrow_2$ is sent corresponds precisely to the length of the "side fuse" multiplied by i.

Without loss of generality, we require that the set of indices I contains only prime numbers (as in Figure 2). Hence, by the unique-prime-factorization theorem, each multiset of numbers in I is uniquely determined by the product of its elements. This leads to a simple verification procedure performed by A_2 at the root: At time 1, node v_ε checks that it receives $\uparrow_1$ and not $\uparrow_2$. After that, it expects to never again see $\uparrow_1$ without $\uparrow_2$, and remains in a loop as long as it gets either no signal at all or both $\uparrow_1$ and $\uparrow_2$. Upon receiving $\uparrow_2$ alone, it exits the loop and verifies that all of its children have sent both signals, which is apparent from the state of each child. The root rejects immediately if any of the expectations above are violated, or if two nodes with different types send the same signal at the same time. Otherwise, it enters an accepting state after leaving the loop. Now, consider the sequence $T = (t_1, \ldots, t_{n+1})$ of rounds in which v_ε receives at least one of the signals $\uparrow_1$ and $\uparrow_2$. It is easy to see by induction on T that successful completion of the procedure above ensures that there is a sequence $S = (i_1, \ldots, i_n)$ of indices in I with the following properties: For each $k \in \{1, \ldots, n\}$, the root has at least one child v_k of type i_k that sends $\uparrow_1$ at time t_k and $\uparrow_2$ at time t_{k+1}, and the "fuse" of v_k encodes precisely the multiset of indices occurring in $(i_1, \ldots, i_{k-1})$. Conversely, each child of v_ε can be associated in the same manner with a unique element of S.

To conclude our proof, we have to argue that the automaton A, which simulates A_1 and A_2 in parallel, accepts some labeled pointed digraph if and only if P has a solution S. The "if" part is immediate, since, by construction, A accepting a ditree-encoding of S is equivalent to S being a valid solution of P. To show the "only if" part, we start with a pointed digraph accepted by A, and incrementally transform it into a ditree-encoding of a solution S, while maintaining acceptance by A: First of all, by Lemma 6, we may suppose that the digraph is a ditree. Its root must be of type ε, since A would not accept otherwise. Next, we require that A raises an alarm at nodes that see an unexpected set of states in their incoming neighborhood, and that this alarm is propagated up to the root, which then reacts by entering a rejecting sink state. This ensures that the repartition of types is consistent with our specification; for example, that the children of a node of type i' must be of type i' themselves. We now prune the ditree in such a way that nodes of type i keep at most two children and nodes of type i' keep at most one child. (The behavior of the deleted children must be indistinguishable from the behavior of the remaining children, since otherwise an alarm would be raised.) This leaves us with a ditree corresponding exactly to the input "expected" by the automaton A_2. Since it is accepted by A_2, this ditree must be very close to an encoding of a solution $S = (i_1, \ldots, i_n)$, with the only difference that each element i_k of S may be represented by several nodes $v_k^1, \ldots, v_k^m$. However, we know by construction that A behaves the same on all of these representatives. We can therefore remove the subtrees rooted at $v_k^2, \ldots, v_k^m$, and thus we obtain a ditree-encoding of S that is accepted by A. □

Acknowledgments

Fabian Reiter wants to thank Olivier Carton for several pleasant discussions and constructive comments. This work is supported by the DeLTA project (ANR-16-CE40-0007).

References

[1] Patrick Blackburn & Johan van Benthem (2007): *Modal logic: a semantic perspective*. In Patrick Blackburn, Johan van Benthem & Frank Wolter, editors: *Handbook of Modal Logic*, *Studies in Logic and Practical Reasoning* 3, Elsevier, pp. 1–84, doi:10.1016/S1570-2464(07)80004-8.

[2] Patrick Blackburn, Maarten de Rijke & Yde Venema (2002): *Modal logic*. *Cambridge Tracts in Theoretical Computer Science* 53, Cambridge University Press, Cambridge, doi:10.1017/CBO9781107050884.

[3] Lauri Hella, Matti Järvisalo, Antti Kuusisto, Juhana Laurinharju, Tuomo Lempiäinen, Kerkko Luosto, Jukka Suomela & Jonni Virtema (2012): *Weak models of distributed computing, with connections to modal logic*. In Darek Kowalski & Alessandro Panconesi, editors: *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, ACM, pp. 185–194, doi:10.1145/2332432.2332466.

[4] Lauri Hella, Matti Järvisalo, Antti Kuusisto, Juhana Laurinharju, Tuomo Lempiäinen, Kerkko Luosto, Jukka Suomela & Jonni Virtema (2015): *Weak models of distributed computing, with connections to modal logic*. *Distributed Computing* 28(1), pp. 31–53, doi:10.1007/s00446-013-0202-3. Available at `https://arxiv.org/abs/1205.2051`.

[5] Neil Immerman (1999): *Descriptive complexity*. Graduate texts in computer science, Springer, doi:10.1007/978-1-4612-0539-5.

[6] Antti Kuusisto (2013): *Modal Logic and Distributed Message Passing Automata*. In Simona Ronchi Della Rocca, editor: *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*, *LIPIcs* 23, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 452–468, doi:10.4230/LIPIcs.CSL.2013.452.

[7] Antti Kuusisto (2014): *Infinite Networks, Halting and Local Algorithms*. In Adriano Peron & Carla Piazza, editors: *Proceedings Fifth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2014, Verona, Italy, September 10-12, 2014.*, *EPTCS* 161, pp. 147–160, doi:10.4204/EPTCS.161.14.

[8] Christof Löding (2012): *Basics on Tree Automata*. In Deepak D'Souza & Priti Shankar, editors: *Modern Applications of Automata Theory*, *IISc Research Monographs Series* 2, World Scientific, pp. 79–109, doi:10.1142/9789814271059_0003.

[9] Fabian Reiter (2015): *Distributed Graph Automata*. In: *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, IEEE Computer Society, pp. 192–201, doi:10.1109/LICS.2015.27. Available at `https://arxiv.org/abs/1408.3030`.

[10] Fabian Reiter (2017): *Asynchronous Distributed Automata: A Characterization of the Modal Mu-Fragment*. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn & Anca Muscholl, editors: *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, *LIPIcs* 80, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 100:1–100:14, doi:10.4230/LIPIcs.ICALP.2017.100. Available at `http://arxiv.org/abs/1611.08554`.

[11] Jukka Suomela (2013): *Survey of local algorithms*. *ACM Comput. Surv.* 45(2), pp. 24:1–24:40, doi:10.1145/2431211.2431223.