




Graph-Walking Automata: From Whence They Come, and Whither They are Bound

Alexander Okhotin^(✉) 

St. Petersburg State University,
7/9 Universitetskaya nab., Saint Petersburg 199034, Russia
alexander.okhotin@spbu.ru

Abstract. Graph-walking automata are finite automata walking on graphs given as an input; tree-walking automata and two-way finite automata are their well-known special cases. Graph-walking automata can be regarded both as a model of navigation in an unknown environment, and as a generic computing device, with the graph as the model of its memory. This paper presents the known results on these automata, ranging from their limitations in traversing graphs, studied already in the 1970s, to the recent work on the logical reversibility of their computations.

1 Introduction

A graph-walking automaton (GWA) walks over a given graph by moving from one node to another along the edges. It is equipped with finite memory, and at each step it uses its current state and the label of the current node to determine its action, that is, which edge to follow and which new state to enter.

A natural prototype for a graph-walking automaton is a robot exploring an unknown environment using limited internal memory and leaving no marks in the environment. A typical task is to traverse the entire graph, for instance, in search for a specially marked node. The most famous example of this task is found in the classical Greek myth about Theseus traversing the Labyrinth and slaying the Minotaur therein.

To be exact, Theseus is faced with three consecutive tasks: first, *finding the Minotaur in the Labyrinth*; second, *slaying him*; and third, *finding the way out*. As the myth is usually told, the main difficulty was to find the way out of the Labyrinth, once the Minotaur is slain. However, as illustrated in Fig. 1, Theseus was fortunate to be helped by Ariadne, whose thread allowed him to return to the gate of the Labyrinth in time $O(n)$. Even though the myth does not explain how Theseus found the Minotaur, it is not difficult to see that even Ariadne's thread alone is sufficient to traverse the entire graph by using an inefficient form of depth-first search.

Supported by the Russian Science Foundation, project 18-11-00100.

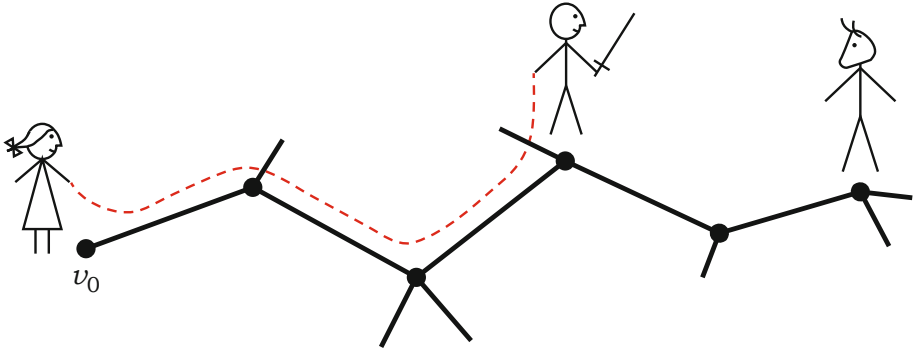


Fig. 1. Theseus searching for the Minotaur in a graph with the help of Ariadne’s thread.

If Theseus were to accomplish his task with no Ariadne to guide him, he would act as a graph-walking automaton. In this case, already the problem of *finding* the Minotaur would present a substantial difficulty. This problem, in general, amounts to traversing the entire graph. The question of whether there exists a finite automaton that can traverse any given undirected graph by following its edges was reportedly first proposed by Michael Rabin: in his 1967 public lecture, he conjectured that this is impossible [12]. The conjecture was confirmed by Budach [9], who proved that for every graph-walking automaton there is a planar graph that it cannot fully traverse. In other words, there is a maze, in which Theseus, without Ariadne, would not even find the Minotaur.

This result by no means contradicts the assumptions made by the Ancient Greeks. Indeed, Theseus had to traverse one particular maze—the Cretan Labyrinth—which was apparently constructed in the way that reaching the Minotaur from the gate was easy, whereas finding the way back was hard. This suggests the following formal representation of the task faced by Theseus if Ariadne turns her back on him: on his own, he begins at the gate to the Labyrinth (the initial node v_0) and, acting as a graph-walking automaton, presumably finds the Minotaur; then, can he return to v_0 while still acting as a graph-walking automaton? Unexpectedly, there is a positive solution to this problem: if there exists a graph-walking automaton that leads Theseus from the Labyrinth gate to the Minotaur, then there is also a graph-walking automaton that leads him back to the gate; this was established by Kunc and Okhotin [19], based on a general idea discovered of Sipser [28]. Roughly speaking, the resulting graph-walking automaton *backtracks all possible paths that lead to the Minotaur according to the method employed by Theseus*.

Besides the direct interpretation of graph traversal as motion in a discrete environment, graph-walking automata also serve as a model of computation. Two simple cases of graph-walking automata are well-known in the literature. First, there are the *two-way deterministic finite automata* (2DFA), which traverse a given input string as a graph that consists of a single path; they are

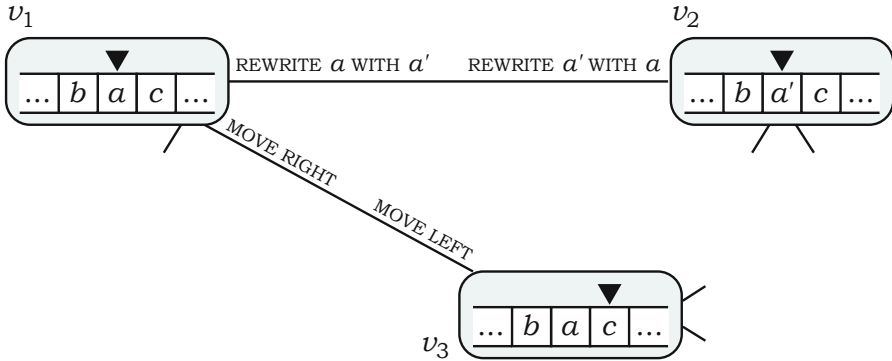


Fig. 2. Memory configurations of a Turing machine modelled by a graph.

notable for being equivalent to one-way finite automata [15, 26], as well as for having a nondeterministic variant that can be determinized, with important complexity issues surrounding the complexity of determinization [16]. The other model are the *tree-walking automata* (TWA), which traverse trees in the same sense as do the graph-walking automata: in contrast to the case of 2DFA, these automata are weaker in power than the “one-way” (bottom-up or top-down) tree automata [8], and their nondeterministic variant is strictly more powerful than the deterministic case [7].

Many other models of computation can be represented as graph-walking automata. The graph represents the memory of a machine: nodes are memory configurations, and edges are elementary operations on the memory. This graph is implicitly constructed for every input object, and then the computation of the machine is interpreted as a walk over this graph. For example, for a Turing machine, nodes correspond to different head positions and tape contents, as illustrated in Fig. 2. If a Turing machine has bac on the tape, with the head at a in a state q , and if it executes a stationary transition that rewrites a with a' and enters a state q' , this corresponds to a GWA at v_1 in the state q , moving to v_2 in the state q' .

This way, many general ideas on computation, such as nondeterminism, reversibility, halting, probabilistic computation, etc., which are defined for various models of computation, admit a unified representation in terms of graph-walking automata. Many standard research problems, such as the complexity of testing membership and emptiness, closure properties and state complexity, can also be represented and studied for graph-walking automata. The particular models of computation can then be regarded as *families of input graphs*, potentially reducing the difference between the models to graph-theoretic properties. This view of graph-walking automata as a powerful general model in automata theory further motivates their study.

This paper gives a brief overview of the graph-walking automaton model. The overview begins with the basic definitions: *graphs* traversed by automata are

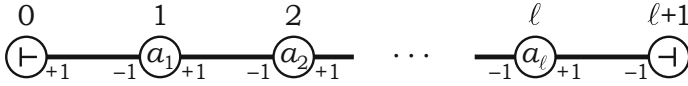


Fig. 3. A string $w = a_1 a_2 \dots a_\ell$ with end-markers, represented as a graph with $V = \{0, 1, 2, \dots, \ell, \ell + 1\}$.

defined in Sect. 2, followed by a definition of graph-walking automata in Sect. 3; justifications for various details of the definitions are provided. Section 4 presents the most well-known result in the area, that there is no graph-walking automaton that can traverse every graph—that is, that Theseus, without Ariadne’s thread, would not find the Minotaur in the Labyrinth. The fundamental construction of a graph-walking automaton that simulates the computations of another graph-walking automaton *backwards* is explained in Sect. 5; by this result, if Theseus finds the Minotaur, then he can find his way back by tracing back his footsteps. The next Sect. 6 describes the applications of this construction to reversible computing: every graph-walking automaton can be transformed to a reversible graph-walking automaton that accepts the same set of finite graphs. Using this result, the closure of graph-walking automata under Boolean operations is established in Sect. 7. Section 8 defines the basic decision problems for graph-walking automata. Possible variants of the graph-walking automaton model are discussed in Sect. 9, whereas algorithms for graph exploration based on other models are briefly mentioned in Sect. 10. The last Sect. 11 suggests some directions for the future study of graph-walking automata.

2 Graphs

A definition of graph-walking automata naturally begins with the form of the graphs on which they walk. There are quite a few details to be fixed: are graphs directed or undirected? finite or infinite? is the initial node specifically marked or not? is the degree of nodes bounded or unbounded? what kind of labels are there to guide the automaton?

Different choices lead to different models, and perhaps, once the theory of graph-walking automata reaches maturity, the difference between the resulting models shall be formally investigated. In the definitions given in the literature, the details were chosen to fit the motivating applications, such as exploring an unknown environment, and representing models of computations in a unified setting.

The relation to other models of computations begins with the simplest case of graph-walking automata: the deterministic two-way finite automata (2DFA). A string, as it is processed by a 2DFA, is the simplest case of a graph. This is a finite connected undirected graph, with its nodes corresponding to positions in the string, and accordingly labelled with input symbols and end-markers (\leftarrow , \rightarrow). The nodes are connected into a chain, as illustrated in Fig. 3. As per the

standard definition of a 2DFA, the automaton moves between the nodes in the directions -1 and $+1$, which are assigned to end-points of edges.

The origins and applications of the graph-walking models researched so far lead to the following choices in the basic definitions.

Graphs are undirected, and every edge can be traversed in both directions. This is a natural assumption under the maze-walking interpretation, where one can always retract the last step. If a graph is taken as a model of a memory, this means that every modification of the memory carried out in a single operation can always be reversed by applying another single operation. In particular, there cannot be a “global reset” operator that erases an unbounded amount of information.

Graphs may, in theory, be infinite, although an accepting computation of a graph-walking automaton still must be a *finite* walk over the input graph. This corresponds to the intuition of traversing a maze, which may be infinite, yet Theseus has to slay only one Minotaur and get back within a finite time. This also fits the definition of classical computation: for instance, even though a Turing machine is equipped with unbounded memory, it must terminate in finitely many steps in order to accept.

However, dealing with infinite graphs in graph-walking automata is usually difficult, and, in particular, all results presented in this paper hold only for finite graphs.

The initial node is specifically marked, that is, upon entering a node, the automaton knows whether it is initial or not. Over this point, there is a certain discrepancy between different applications. On the one hand, the initial node is the gate to the Labyrinth, where Theseus begins his journey, and where to he expects to get back after slaying the Minotaur: of course, upon visiting a node, it should be immediately visible whether it is the gate.

On the other hand, a marked initial node in a computing device means that the machine always knows whether its memory is in the initial configuration. This holds for simpler kinds of automata, such as 2DFA, which can see whether their heads are in the initial position. However, for a Turing machine, this means that at any moment it should know whether its entire work-tape is clear; in practice, this would require re-scanning the tape.

Some of the existing results on graph-walking automata assume graphs with a marked initial node, and some results refer to the case of an unmarked initial node.

The degree of nodes is bounded by a constant. Furthermore, the end-points of edges meeting at each node are labelled with different *directions* from a fixed finite set D . This is necessary for an automaton to distinguish between these edges, and to be able to proceed in each available direction.

In a maze, this means that a bounded number of corridors meet at every junction, and that each outgoing corridor has a unique label. For a computing

device, directions are elementary operations on the memory, and following an edge means applying that operation.

In order to handle graphs of unbounded degree, nodes of higher degrees can be split into subgraphs, and so this restriction is actually inessential.

Nodes are labelled, and so are the end-points of edges. Node labels are analogous to symbols in a string, and the set of possible labels is accordingly denoted by Σ . At every moment of its computation, a graph-walking automaton can observe only the label of the current node v , denoted by $a = \lambda(v)$. Furthermore, the label of a node determines the set of directions available in that node, denoted by D_a , with $D_a \subseteq D$. Knowing this label and using its internal state, the automaton decides, in which direction from D_a to proceed, and which state to enter.

In a string, there are two directions: to the previous symbol (-1) and to the next symbol ($+1$), with $D = \{-1, +1\}$. It is essential that moving to the direction $+1$ and then to the direction -1 always leads back to the same symbol: that is, $+1$ and -1 are *opposite directions*. In the general case of graphs, every direction d in D must have an opposite direction $-d \in D$, defined by a bijective operator $- : D \rightarrow D$.

*With the above details fixed, graphs processed by a graph-walking automaton are defined over a **signature***, which includes a set of directions, a set of node labels, and a set of available directions for each node label, as well as identifies the labels for the initial node. A signature is generalization of an alphabet for the case of graphs.

Definition 1 (Kunc and Okhotin [19]). *A signature is a quintuple $\mathcal{S} = (D, -, \Sigma, \Sigma_0, \langle D_a \rangle_{a \in \Sigma})$, where*

- D is a finite set of directions, that is, labels at the end-points of edges;
- $- : D \rightarrow D$ is a bijective operator that defines the opposite direction, it satisfies $-(-d) = d$ for all $d \in D$;
- Σ is a finite set of node labels;
- $\Sigma_0 \subseteq \Sigma$ is a non-empty subset of labels allowed in the initial node, whereas all other nodes have to be labelled with elements of $\Sigma \setminus \Sigma_0$;
- each $D_a \subseteq D$, with $a \in \Sigma$, is the set of directions available in all nodes labelled with a , so that every such node must have degree $|D_a|$, with the incident edges corresponding to the elements of D_a .

Graphs over a signature \mathcal{S} are undirected labelled graphs defined as follows.

Definition 2 (Kunc and Okhotin [19]). *A graph over a signature $\mathcal{S} = (D, -, \Sigma, \Sigma_0, \langle D_a \rangle_{a \in \Sigma})$ is a quadruple $(V, v_0, +, \lambda)$, where*

- V is a set of nodes;
- $v_0 \in V$ is the initial node;
- $\lambda : V \rightarrow \Sigma$ is a function assigning a label to each node $v \in V$, so that the label $\lambda(v)$ is in Σ_0 if and only if $v = v_0$;

- $+$: $V \times D \rightarrow V$ is a function representing the edges of the graph: it is defined in each node $v \in V$ and for each direction $d \in D_{\lambda(v)}$ applicable in that node, so that the neighbour of v in the direction d is denoted by $v + d$.
The neighbour of v in the direction $-d$ is accordingly defined by $v - d$. The graph must satisfy the condition $(v + d) - d = v$, for all $v \in V$ and $d \in D_{\lambda(v)}$. In particular, $D_{\lambda(v+d)}$ must have the direction $-d$.

A graph with an unmarked initial node is defined in the same way, but with no special label for v_0 : every node $v \in V$ must have $\lambda(v) \notin \Sigma_0$.

Example 1. Strings over an alphabet Γ delimited by left and right end-markers (\vdash , \dashv) are represented as graphs over a signature $\mathcal{S} = (D, -, \Sigma, \Sigma_0, \langle D_a \rangle_{a \in \Sigma})$ with directions $D = \{+1, -1\}$, where $-(+1) = -1$, and with node labels $\Sigma = \Gamma \cup \{\vdash, \dashv\}$. The only initial label is the left end-marker: $\Sigma_0 = \{\vdash\}$. The set of directions at each input symbol $a \in \Gamma$ is $D_a = \{+1, -1\}$. Only one direction available at each end-marker: $D_{\vdash} = \{+1\}$, $D_{\dashv} = \{-1\}$.

Every connected graph over the signature \mathcal{S} is a labelled path graph of the form depicted in Fig. 3. It corresponds to a string over Γ .

3 Automata

Definition 3 (Kunc and Okhotin [19]). A deterministic graph-walking automaton over a signature $\mathcal{S} = (D, -, \Sigma, \Sigma_0, \langle D_a \rangle_{a \in \Sigma})$ is a quadruple $\mathcal{A} = (\mathcal{S}, Q, q_0, \delta, F)$, in which

- Q is a finite set of states;
- $q_0 \in Q$ is the initial state;
- $F \subseteq Q \times \Sigma$ is a set of acceptance conditions;
- $\delta: (Q \times \Sigma) \setminus F \rightarrow Q \times D$ is a partial transition function, with $\delta(q, a) \in Q \times D_a$ for all a and q where it is defined.

The automaton gets a graph $(V, v_0, +, \lambda)$ over the signature \mathcal{S} as an input. At each point of its computation, the automaton is at a node $v \in V$ in a state $q \in Q$; the pair (q, v) is known as the automaton's *configuration*. The initial configuration is (q_0, v_0) , that is, the automaton begins at the initial node in its initial state. At each step of the computation, while in a configuration (q, v) , the automaton observes the symbol $\lambda(v)$ and evaluates its transition function on $\delta(q, \lambda(v))$. There are three possibilities.

- If this value is defined, let $\delta(q, \lambda(v)) = (q', d)$. Then the automaton moves in the direction d and enters the state q' , so that the next configuration is $(q', v + d)$.
- If δ is undefined on $(q, \lambda(v))$, then the automaton halts. If $(q, \lambda(v)) \in F$, it accepts the input graph, and if $(q, \lambda(v)) \notin F$, it rejects.

The computation on a given graph is uniquely defined, and it can either be infinite, or accepting, or rejecting. The set of graphs recognized by the automaton \mathcal{A} consists of all graphs over the signature \mathcal{S} on which it halts and accepts.

Example 2. Let $\mathcal{S} = (D, -, \Sigma, \Sigma_0, \langle D_a \rangle_{a \in \Sigma})$ be a signature for strings over an alphabet Γ represented as graphs, as defined in Example 1. A graph-walking automaton over this signature is a deterministic two-way finite automaton (2DFA).

Another well-known special case of graph-walking automata are the *tree-walking automata* operating on trees of a bounded degree. This model was first defined by Aho and Ullman [1, Sect. VI], and later Bojańczyk and Colcombet [7, 8] showed that these automata are weaker than bottom-up and top-down tree automata, and that their nondeterministic variant cannot be determinized. Since a string is a special case of a tree with out-degree 1, the signature for trees processed by these automata generalizes the one from Example 1 by providing additional directions.

Example 3. Trees of degree k with nodes labelled by symbols from a set Γ are defined over a signature $\mathcal{S} = (D, -, \Sigma, \Sigma_0, \langle D_a \rangle_{a \in \Sigma})$, with the set of directions $D = \{+1, \dots, +k, -1, \dots, -k\}$, where each direction $+i$ means going down to the i -th successor, whereas $-i$ points from the i -th successor to its predecessor. The directions $+i$ and $-i$ are opposite: $-(+i) = -i$.

Nodes are labelled with the symbols from a set $\Sigma = \{\top, \perp_1, \dots, \perp_k\} \cup (\Gamma \times \{1, \dots, k\})$. The root node v_0 is labelled by the top marker (\top), with $D_\top = \{+1\}$ and $\Sigma_0 = \{\top\}$. Each i -th bottom marker (\perp_i) has $D_{\perp_i} = \{-i\}$, and serves as a label for leaves. Internal nodes are labelled with elements of $\Gamma \times \{1, \dots, k\}$, so that a label (a, i) , with $a \in \Gamma$ and $i \in \{1, \dots, k\}$, indicates a node containing a symbol a , which is the i -th successor of its predecessor; the set of available directions is $D_{(a,i)} = \{-i, +1, \dots, +k\}$.

Connected graphs over this signature are exactly the k -ary trees augmented with a top marker and with bottom markers.

4 To Find the Minotaur

In terms of graph-walking automata, the problem of maze exploration is represented as follows. In the signature, the alphabet Σ contains two special labels, one marking the initial node, and the other marking the location of the Minotaur. A graph-walking automaton has to test whether the given Labyrinth contains at least one Minotaur, as illustrated in Fig. 4.

This problem is often stated for graphs with an unmarked initial node: in other words, Theseus appears in the middle of the Labyrinth and has to test whether there is at least one Minotaur in the Labyrinth. In this setting, there is the following well-known result.

Theorem 1 (Budach [9]). *There exists a signature \mathcal{S} , such that for every graph-walking automaton \mathcal{A} over \mathcal{S} there is a planar graph G over \mathcal{S} , with an unmarked initial node, such that the computation of \mathcal{A} of G does not visit one of its nodes.*

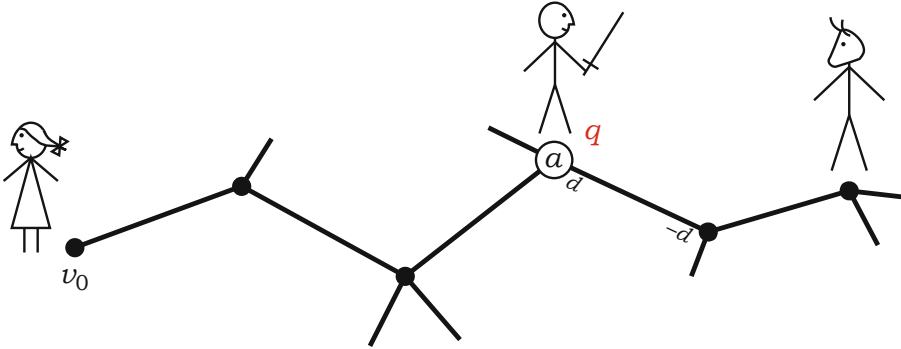


Fig. 4. Theseus using finite memory and node and edge labels to search for the Minotaur without Ariadne’s thread.

The original proof of this result was very sophisticated. Later, the following short and clear proof was discovered.

Lemma 1 (Fraigniaud et al. [12]). *For every $d \geq 3$, let the signature $\mathcal{S}_d = (D, -, \Sigma, \Sigma_0, \langle D_a \rangle_{a \in \Sigma})$ have $D = \{1, \dots, d\}$, with $-i = i$ for all $i \in D$, $\Sigma = \{a\}$, $\Sigma_0 = \emptyset$ and $D_a = D$. Then, for every n -state graph-walking automaton \mathcal{A} over \mathcal{S} there is a planar connected graph $G = (V, v_0, +, \lambda)$ over \mathcal{S} with unmarked initial node and with $|V| \leq n + d + 3$, on which \mathcal{A} does not visit all nodes.*

Proof (a sketch). Since $\Sigma = \{a\}$, all nodes of the graph appear identical, and the automaton’s transitions by a induce a sequence of states q_0, q_1, \dots , and a sequence of directions d_1, d_2, \dots , with $\delta(q_i, a) = (q_{i+1}, d_{i+1})$. The sequence of states is periodic with some period p , with $p \in \{1, \dots, n\}$, so that $q_{i+p} = q_i$ for all $i \geq n - 1$. Therefore, the sequence of directions has the same period.

At first, it is convenient to assume that the automaton operates on an infinite tree. Then, the periodic part of the sequence of directions can either drive the automaton into a cycle, or set it into periodic motion leading away from the initial node. In the former case, the automaton actually visits only finitely many nodes of the tree; it is sufficient to take the resulting subtree with one extra node, and to reconnect the unused edges between these nodes.

In case the automaton moves away along a periodic sequence of directions, the general idea is to *merge two nodes* of the infinite tree that are visited in the same state, thus effectively replacing this tree with a finite “trap”, on which the automaton follows the same periodic trajectory. Consider the example in Fig. 5(top), where the sequence of directions is $a(bccabacab)^\omega$, with the same state q visited after each prefix in $a(bccabacab)^*$. The periodic part contains a detour cc , and with this detour omitted, it takes the form $babacab$. Let periodic part of the sequence, with all detours removed, be of the form $\alpha\beta\alpha^R$, where $\alpha, \beta \in D^*$ and α^R denotes the reversal of α : in Fig. 5(top), $\alpha = ba$ and $\beta = bac$. The plan is to join the nodes on the border of β , so that the periodic part visits α twice. The resulting trap is given in Fig. 5(bottom left).

It remains to add one extra node and fill in the missing nodes, as done in Fig. 5(bottom right).

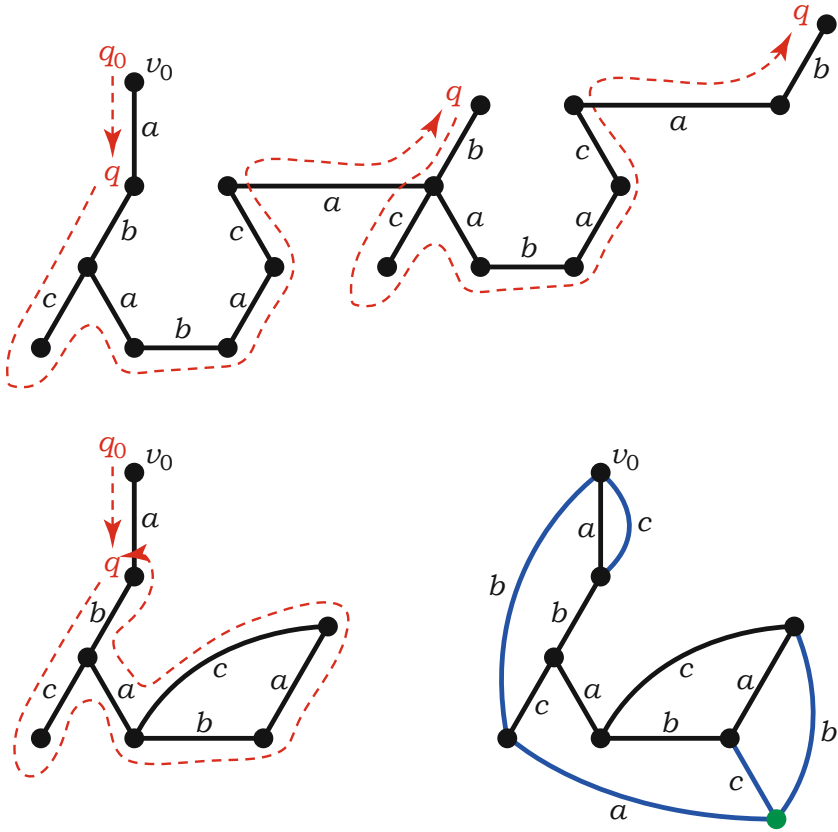


Fig. 5. Construction of a trap in the proof of Lemma 1: (top) A periodic sequence on an infinite tree; (bottom left) A trap constructed by merging two nodes in the periodic part; (bottom right) The trap with an unreachable node and all missing transitions added.

The full proof has several complicated cases, such as the case when the periodic part with detours removed is an odd palindrome. However, all cases are handled by a similar, yet more sophisticated construction, with the periodic computation condensed into looping in a finite trap. \square

The above proof does not directly apply to the case of graph-walking automata with a marked initial node, because every appearance of v_0 in various places of the sequence disrupts the argument. One can likely work around these issues by an extensive case analysis. A concise new proof of this result for the marked case has been found by Martynova [22], and shall be published in due time.

Since the problem of graph exploration is an important algorithmic problem, if graph-walking automata cannot handle it in general, then the question is, which models can? For instance, if graph-walking automata were equipped

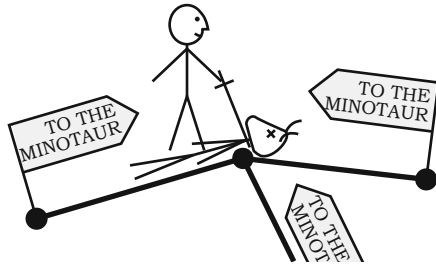
with finitely many pebbles that can be dropped at any nodes and later picked up, then could there exist an automaton of this kind that can traverse every maze? The answer is again negative; as shown by Rollik [27], even a *team of communicating automata* cannot traverse all graphs (a team may be regarded as a single automaton equipped with multiple heads: at every moment each head is stationed at some node, and the automaton observes the symbols in those nodes, as well as senses whether any heads share the same position).

An interesting special case studied in the literature is the case of embedded plane graphs, with all edges going either from north to south or from east to west, and with the automaton being aware of the direction of each edge. As proved by Blum and Kozen [6], there is a graph-walking automaton equipped with *two pebbles* that can walk through every such maze.

5 Tracing Back the Footsteps

Several results on graph-walking automata are based on the same common construction of an automaton that traces back all computations of a given automaton that lead to a particular configuration.

Suppose Theseus stands over the fresh corpse of the Minotaur, remembering nothing besides the finite-state transition rules he has followed to reach this place. Can he trace back his own footsteps and find the exit? This is not obvious at all: for instance, if every node in the Labyrinth is equipped with a plaque “to the Minotaur” pointing to the



most suitable edge, then the Minotaur can be found using a one-state transition table. However, such plaques would not help getting back: as illustrated in the picture, Theseus would not even know the penultimate node on his path to the Minotaur.

Nevertheless, the answer to the question is positive, and Theseus can always find the way back by tracing back all the paths that would lead him to the Minotaur according to his own transition rules. This is done using the general idea of *backtracking the tree of accepting computations*, discovered by Sipser [28] in his study of halting space-bounded computations. This idea has been reused many times: for instance, Lange et al. [21] applied it in their proof of the equivalence of deterministic space $O(s(\ell))$ to reversible space $O(s(\ell))$. Kondacs and Watrous [17] have improved the details of Sipser’s construction, leading to a simulation of an n -state 1DFA by a reversible 2DFA with as few as $2n$ states. Geffert et al. [14], Muscholl et al. [24] and Morita [23] used the same idea to produce similar constructions for various automaton models, with various degree of efficiency.

Actually, this construction applies to particular automaton models so easily, for the reason that it is *correct in the general case of graph-walking automata*—and therefore it holds true for their particular cases, such as all automaton models to which it was previously applied.

In order to define the construction for tracing back the footsteps of a graph-walking automaton, the following property of automata turns out to be useful. A GWA is said to be *direction-determinate*, if it always remembers the direction, in which it came to the current node.

Definition 4. A graph-walking automaton $\mathcal{A} = (\mathcal{S}, Q, q_0, \delta, F)$ over a signature $\mathcal{S} = (D, -, \Sigma, \Sigma_0, \langle D_a \rangle_{a \in \Sigma})$, is direction-determinate, if, for some partial function $d: Q \rightarrow D$, whenever a transition $\delta(p, a)$ leads the automaton to a state q , it must move in the direction $d(q)$.

Then, for each $a \in \Sigma$, the transitions by a are defined by a partial function $\delta_a: Q \rightarrow Q$, with $\delta(p, a) = (q, d(q))$, where $q = \delta_a(p)$.

Every graph-walking automaton with a set of states Q can be transformed to a direction-determinate automaton with the set of states $Q \times D$, which simply remembers the last direction it has taken in the second component of its state, without ever using this information.

Lemma 2 (Kunc and Okhotin [19]). For every direction-determinate graph-walking automaton $\mathcal{A} = (\mathcal{S}, Q, q_0, \delta, F)$, there exists an automaton over the same signature \mathcal{S} and with the set of states $\overrightarrow{Q} \cup [Q]$, where $\overrightarrow{Q} = \{ \overrightarrow{q} \mid q \in Q \}$ and $[Q] = \{ [q] \mid q \in Q \}$, which, on any finite graph, backtracks all computations of \mathcal{A} leading to any accepting configuration (\hat{q}, \hat{v}) , with $(\hat{q}, \lambda(\hat{v})) \in F$, in the following sense: if \mathcal{B} begins its computation in the configuration $([\hat{q}], \hat{v} - d(\hat{q}))$, then it passes through all such configurations $([q], v)$, that the computation of \mathcal{A} beginning in $(q, v + d(q))$ accepts in the configuration (\hat{q}, \hat{v}) .

Proof (a sketch). The automaton \mathcal{B} traverses the tree of all computations of \mathcal{A} that terminate in the configuration (\hat{q}, \hat{v}) . The construction is based on ordering the states in Q , which implies an ordering on the branches of the computation tree. Then, whenever \mathcal{B} finds a configuration of \mathcal{A} that has some predecessors, it proceeds with tracing back the *least* of the computation paths leading to the current configuration. Once \mathcal{B} reaches a configuration of \mathcal{A} without predecessors, it switches to forward simulation, which is carried out in the states \overrightarrow{Q} . In these “forward states”, for every configuration of \mathcal{A} visited by \mathcal{B} , it tries to trace back the *next* computation path according to the chosen ordering. If the path being traced forward is already greater than the other paths meeting at the present point, then the forward simulation continues. This way, the entire tree of computations converging in the chosen configuration is eventually traversed. \square

Sipser’s [28] paper actually contains two methods for implementing this kind of computation tree traversal: his first method involves *remembering two states*

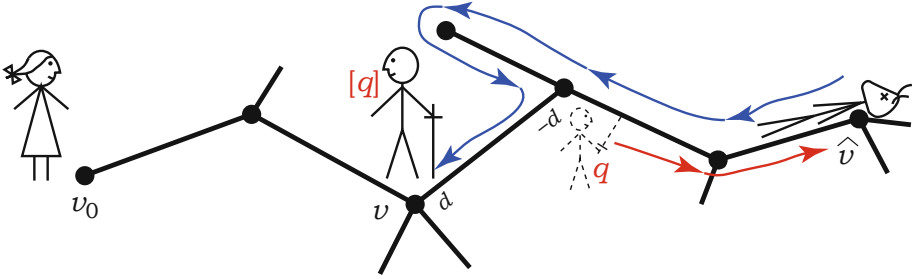


Fig. 6. Theseus tracing back his path from the gate to the Minotaur: on the way back, he is in the configuration $([q], v)$ if, on his way forward, he would reach the Minotaur from the configuration $(q, v + d(q))$.

of the original automaton, that is, produces the set of states $Q \times Q$; the second method sketched by Sipser [28] requires remembering a state and a symbol, that is, uses the states $Q \times \Sigma$. The improvement contributed by Kondacs and Watrous [17] was to remember a state and one bit, using states $Q \times \{0, 1\}$. The construction in Lemma 2 implements the method of Kondacs and Watrous [17] for graph-walking automata, which seem to be the natural limit of the applicability of this method.

Lemma 2 directly implies the promised result that if Theseus can find the Minotaur using finite-state transition rules, then he can get back, as illustrated in Fig. 6. In terms of graph-walking automata, this is formalized as follows.

Definition 5. A graph-walking automaton $\mathcal{A} = (\mathcal{S}, Q, q_0, \delta, F)$ is called returning, if it can accept only at the initial node, that is, $F \subseteq Q \times \Sigma_0$.

Theorem 2 (Kunc and Okhotin [19]). For every n -state graph-walking automaton \mathcal{A} over some signature $\mathcal{S} = (D, -, \Sigma, \Sigma_0, \langle D_a \rangle_{a \in \Sigma})$, with $|D| = d$, there exists a direction-determinate returning graph-walking automaton with $3dn$ states that accepts the same set of finite graphs.

Proof (a sketch). The given GWA \mathcal{A} is first transformed to a dn -state direction-determinate GWA \mathcal{B} . By Lemma 2, there is a $2dn$ -state direction-determinate automaton \mathcal{C} that traces back all accepting computations of \mathcal{B} . The promised returning automaton operates as follows: first, it behaves as \mathcal{B} until it reaches an accepting configuration; then, it behaves as \mathcal{C} , accepting if it ever encounters the initial configuration of \mathcal{B} , and rejecting if it ever returns to its accepting configuration.

Remark 1 (Martynova [22]). If the resulting returning graph-walking automaton is not required to be direction-determinate, then it is sufficient to use only $(2d + 1)n$ states. The new automaton first behaves as \mathcal{A} until acceptance, and then proceeds with simulating \mathcal{C} .

The property of being returning is crucial for Theseus, and it can also be useful in other exploration problems. On the other hand, as a property of computing

devices, it is not always relevant: for instance, if a Turing machine is represented as a graph-walking automaton, then being returning only means that it always restores the original contents of its tapes before acceptance.

As it shall now be demonstrated, the construction for tracing back the accepting computations can ensure several important general properties of abstract machines.

6 Reversible Computation

Reversibility is a stronger form of determinism: a deterministic machine is reversible, if its computations are also backward deterministic, that is, given its current configuration, its configuration at the previous step can always be uniquely determined. This is an interesting theoretical notion.

Furthermore, reversibility of computations is important from the point of view of the physics of computation [5]: according to *Landauer's principle* [20], a logically irreversible erasing of one bit of information incurs dissipation of $kT \ln 2$ joules of energy, where k is the Boltzmann constant and T is the temperature. Reversible computing, in theory, can avoid this effect. Furthermore, since the laws of quantum mechanics are reversible, reversible computing is the basic case of quantum computing. These applications further motivate theoretical studies on reversible computation.

In automata theory, Kondacs and Watrous [17] showed that reversible 2DFA can recognize every regular language, whereas reversible 1DFA, as well as one-way quantum automata (1QFA) recognize a proper subset of regular languages. Every Turing machine can be simulated by a reversible Turing machine [5]. Lange et al. [21] proved that the class of sets recognized by reversible Turing machines in space $O(s(n))$, denoted by $\text{RSPACE}(s(n))$, is equal to $\text{DSPACE}(s(n))$. This was done by applying Sipser's [28] method for tracing back accepting computations. The same method was used a few times for different automaton models, showing that their reversible subclass is as powerful as the full deterministic class. This general argument again works for graph-walking automata, and can be established in a clearer form for this model.

Definition 6 (Kunc and Okhotin [19]). *A graph-walking automaton $\mathcal{A} = (\mathcal{S}, Q, q_0, \delta, F)$ is called reversible, if it is direction-determinate and returning, every partial transformation δ_a is injective, and for each initial label $a_0 \in \Sigma_0$, there is at most one state q with $(q, a_0) \in F$.*

To put it simple, reversibility means that every computation can equally be executed backwards. In particular, given a finite graph with $\lambda(v_0) = a_0$, which can only be accepted in the configuration (q, v_0) , one can proceed with backtracking a potential accepting computation beginning in this configuration. Since all partial transformations are injective, the predecessor configuration shall always be uniquely defined, until one of the following two outcomes: either the simulation backtracks to the initial configuration (q_0, v_0) , or an unreachable configuration is reached. In the latter case, the automaton rejects this graph.

The automaton constructed in Lemma 2 is actually reversible.

There is also a notion of a *strongly reversible* graph-walking automaton, satisfying some further requirements: in particular, if a graph is rejected, then it can be rejected only in the initial node, in a particular configuration (q_{rej}, v_0) ; furthermore, all transformations δ_a with $a \notin \Sigma_0$ must be bijective. As long as the input graph is finite, this forces the automaton to halt on every input in one of the two predetermined configurations at the initial node.

Theorem 3 (Kunc and Okhotin [19]). *For every n -state direction-determinate returning graph-walking automaton, there exists a strongly reversible graph-walking automaton with $2n + 1$ states recognizing the same set of finite graphs.*

Proof (a sketch). First, Lemma 2 is applied to the given returning automaton \mathcal{A} . The resulting automaton is augmented with a new initial state, in which it initiates tracing back the computations of \mathcal{A} leading to its accepting configuration (owing to the fact that all accepting configurations of \mathcal{A} are at the initial node). \square

Corollary 1. *For every n -state graph-walking automaton over a signature with d directions, there is a strongly reversible automaton with $6dn + 1$ states that recognizes the same set of finite graphs.*

The notion of a reversible automaton applies to infinite graphs as well. Unfortunately, the above construction of a reversible automaton is apparently useless in that case. Indeed, on an infinite graph, the constructed reversible GWA may, and typically will, immediately rush away from the initial node along an infinite path, perfectly reversibly and perfectly uselessly.

7 Closure Properties and State Complexity

In the case of string languages, a plenty of regularity-preserving operations are known. The study of how they affect the number of states in finite automata is known as the *state complexity*. The state complexity of all reasonable operations on 1DFA and 1NFA is known perfectly well. For unambiguous automata (1UFA) and for two-way automata (2DFA, 2NFA), there are only partial results, and many interesting problems are still open.

Closure properties of graph-walking automata are a new subject, and, at the moment, there are hardly any operations to be studied besides the Boolean operations. The existing results on graph-walking automata easily imply their closure under all Boolean operations.

Theorem 4. *Let \mathcal{A} and \mathcal{B} be two graph-walking automata, with m and n states, respectively, defined over the same signature $\mathcal{S} = (D, -, \Sigma, \Sigma_0, \langle D_a \rangle_{a \in \Sigma})$ with $|D| = d$. Then, there exist:*

1. *a graph-walking automaton \mathcal{C} with $6dm + n + 1$ states that accepts a finite graph if and only if it is in the union $L(\mathcal{A}) \cup L(\mathcal{B})$;*

2. a graph-walking automaton \mathcal{D} with $(2d+1)m+n$ states that accepts a finite graph if and only if it is in the intersection $L(\mathcal{A}) \cap L(\mathcal{B})$;
3. a graph-walking automaton \mathcal{E} with $6dm+1$ states that accepts a finite graph if and only if it is in the complement $\overline{L(\mathcal{A})}$.

Proof (a sketch). (Union) To begin with, the first automaton \mathcal{A} is made strongly reversible, so that it halts on every input at the initial node. Then the automaton \mathcal{C} for the union simulates the \mathcal{A} , and accepts if it accepts, or proceeds with simulating \mathcal{B} if it rejects.

(Intersection) By Theorem 2 and Remark 1, the first automaton is made returning. The automaton \mathcal{D} first simulates it until it returns and is about to accept; if this happens, then it turns to simulating \mathcal{B} .

(Complementation) It is sufficient to make the automaton strongly reversible, and then exchange the acceptance and rejection decisions. \square

8 Decision Problems

The most commonly studied decision problems on automata include the membership problem (whether a given automaton accepts a given input); the emptiness problem (whether a given automaton accepts any input); the universality problem (whether a given automaton accepts every input); the equivalence problem (whether two given automata accept exactly the same inputs); and the inclusion problem (whether every input accepted by the first of the two given automata is accepted by the other). In each case, the “input” is taken to be any graph over the same signature as the automaton.

The membership problem for graph-walking automata is obviously decidable in time $O(mn)$, where n is the size of the automaton and m is the size of the input. One simply has to run the automaton until it accepts or rejects, or until mn steps have been made, after which it can be pronounced looped.

The rest of the problems (emptiness, universality, equivalence, inclusion) are reducible to each other in view of the closure under all Boolean operations. Whether these problems are decidable, remains unknown. If “emptiness” is interpreted as the existence of a graph accepted by the automaton that belongs to a certain family, then, for certain families of graphs, such as for all grids, the problem becomes undecidable [29]. However, since a graph-walking automaton cannot ensure that a given graph is a grid, this undecidability is hardly relevant to the properties of automata. For the emptiness problem in the form “does there exist any graph over the signature that is accepted by the automaton?”, it is unknown whether it is decidable or not.

In the special cases of 2DFA and TWA, the same problems are well-known to be decidable. One way of showing this for 2DFA is to define the *behaviour function* of the given automaton on substrings, which specifies the outcome of a computation entering a substring from the left or from the right in every possible state. Since there are finitely many such functions, one can determine the set of functions that are actually implemented on some substring. This is sufficient to

decide whether any string is accepted. For TWA over k -ary trees, the algorithm is the same, using subtrees instead of substrings, and dealing with computations enterable from $k + 1$ different sides. Here the number of behaviour functions is still finite, leading to the same enumeration of functions implemented on some trees.

The problem with graphs is that, for graphs of an unrestricted form, it seems impossible to have an upper bound on the number of entry points to subgraphs. Then, a full enumeration of implementable behaviour functions cannot be done, and the argument breaks down.

9 Variants of Graph-Walking Automata

This paper concentrates on one particular kind of graph-walking automata, as explained and justified in Sects. 2 and 3. What kind of other related models could be considered?

First, one can consider different definitions of graphs. Graphs may be *directed*, in the sense that some edges can be passed only in one direction. An example of a GWA operating on a directed graph is a 1DFA, in which the graph is a directed chain. Could any interesting results be obtained along these lines?

The case of infinite graphs is definitely interesting, yet all the constructions presented in this paper are valid only for finite graphs. Some new ideas are needed to handle the infinite case.

The case of an unmarked initial node does not look much different from the marked case. However, all results related to reversibility rely on the initial node's being marked. It remains to investigate whether this is really essential or not.

Leaving the form of the graphs alone, it is interesting to consider the standard modes of computation for graph-walking automata. Most models of computations, from one-way finite automata to Turing machines, have all kinds of variants: reversible, deterministic, nondeterministic, unambiguous, probabilistic, alternating, etc. These definitions apply to graph-walking automata as well. What kind of results could one expect?

As proved by Bojańczyk and Colcombet [7], nondeterministic tree-walking automata cannot be determinized, and so this holds for graph-walking automata in general. However, graph-walking automata over some signatures *can* be determinized, and, for instance, it would be interesting to characterize those signatures. The unambiguous mode of computation is yet to be investigated even for the tree-walking automata.

A kind of one-way model for graphs, the *tilings on a graph*, was considered by Thomas [29]. A tiling assigns labels to all nodes of the graph, so that the labellings of all small subgraphs satisfy certain given conditions. A deterministic graph-walking automaton, and even a nondeterministic one, can be simulated by a tiling. In the case of trees, tilings are bottom-up tree automata, and, by the result of Bojańczyk and Colcombet [7], tree-walking automata are weaker than tilings. This extends to graph-walking automata, yet some special cases are worth being considered.

10 Graph Exploration Algorithms

Since there is no GWA that can traverse any graph, this means that searching in the graph requires devices or algorithms that either use internal memory of more than constant size, or store any information in the nodes of the graph, or do both. In these terms, Ariadne's thread can be regarded as information stored in the nodes. The standard depth-first search includes both Ariadne's thread (as the stack) and marks left in the visited nodes.

Many new algorithms for graph exploration are being developed. For instance, Disser et al. [10] presented a solution using $O(\log \log n)$ pebbles and $O(\log n)$ internal states; this is one of the several algorithms obtained by derandomizing randomized algorithms for graph exploration, which were first investigated by Aleliunas et al. [3]. Algorithms by Albers and Henzinger [2], and by Panaite and Pelc [25] are aimed to minimize the number of edge traversals. Algorithms for searching in a graph with an unbounded degree of nodes have recently been presented by Asano et al. [4] and by Elmasry et al. [11].

From the point of view of automata theory, the question is: can the graph-walking automata be somehow extended to contribute to the more practical methods of graph exploration?

11 Conclusion

The study of graph-walking automata looks like a promising direction in automata theory, yet, at the moment, it is still in its infancy. There are only a few isolated results, which are useful for representing the general form of known generic ideas, but insufficient to form a theory. On the other hand, there are plenty of uninvestigated basic properties, and some of them may turn out relatively easy to determine.

A suggested possible starting point for research is finding a new special case of graphs, along with a motivation for considering it, and then investigate its basic properties. It would be particularly fortunate to find an intrinsically interesting simple case: for instance, for 2DFA, their simple case is the case of a unary alphabet [13, 14, 18], for which much more is known than for 2DFA over multiple-symbol alphabets. Is there such a non-trivial class of non-path graphs that could similarly drive the early research on graph-walking automata?

References

1. Aho, A.V., Ullman, J.D.: Translations on a context free grammar. *Inf. Control* **19**(5), 439–475 (1971)
2. Albers, S., Henzinger, M.R.: Exploring unknown environments. *SIAM J. Comput.* **29**(4), 1164–1188 (2000). <https://doi.org/10.1137/S009753979732428X>
3. Aleliunas, R., Karp, R.M., Lipton, R.J., Lovász, L., Rackoff, C.: Random walks, universal traversal sequences, and the complexity of maze problems. In: *Proceedings of 20th Annual Symposium on Foundations of Computer Science, FOCS 1979*, pp. 218–223. IEEE Computer Society (1979). <https://doi.org/10.1109/SFCS.1979.34>

4. Asano, T., et al.: Depth-first search using $O(n)$ bits. In: Ahn, H.-K., Shin, C.-S. (eds.) ISAAC 2014. LNCS, vol. 8889, pp. 553–564. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13075-0_44
5. Bennett, C.H.: The thermodynamics of computation—a review. *Int. J. Theor. Phys.* **21**(12), 905–940 (1982). <https://doi.org/10.1007/BF02084158>
6. Blum, M., Kozen, D.: On the power of the compass (or, why mazes are easier to search than graphs). In: Proceedings of 19th Annual Symposium on Foundations of Computer Science, FOCS 1978, pp. 132–142. IEEE Computer Society (1978). <https://doi.org/10.1109/SFCS.1978.30>
7. Bojańczyk, M., Colcombet, T.: Tree-walking automata cannot be determinized. *Theor. Comput. Sci.* **350**(2–3), 164–173 (2006). <https://doi.org/10.1016/j.tcs.2005.10.031>
8. Bojańczyk, M., Colcombet, T.: Tree-walking automata do not recognize all regular languages. *SIAM J. Comput.* **38**(2), 658–701 (2008). <https://doi.org/10.1137/050645427>
9. Budach, L.: Automata and labyrinths. *Math. Nachr.* **86**(1), 195–282 (1978). <https://doi.org/10.1002/mana.19780860120>
10. Dissler, Y., Hackfeld, J., Klimm, M.: Undirected graph exploration with $O(\log \log n)$ pebbles. In: Krauthgamer, R. (ed.) Proceedings of 27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, pp. 25–39. SIAM (2016). <https://doi.org/10.1137/1.9781611974331.ch3>
11. Elmasry, A., Hagerup, T., Kammer, F.: Space-efficient basic graph algorithms. In: Mayr, E.W., Ollinger, N. (eds.) Proceedings of 32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015. LIPIcs, vol. 30, pp. 288–301. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015). <https://doi.org/10.4230/LIPIcs.STACS.2015.288>
12. Fraigniaud, P., Ilcinkas, D., Peer, G., Pelc, A., Peleg, D.: Graph exploration by a finite automaton. *Theoret. Comput. Sci.* **345**(2–3), 331–344 (2005). <https://doi.org/10.1016/j.tcs.2005.07.014>
13. Geffert, V., Mereghetti, C., Pighizzini, G.: Converting two-way nondeterministic unary automata into simpler automata. *Theoret. Comput. Sci.* **295**, 189–203 (2003). [https://doi.org/10.1016/S0304-3975\(02\)00403-6](https://doi.org/10.1016/S0304-3975(02)00403-6)
14. Geffert, V., Mereghetti, C., Pighizzini, G.: Complementing two-way finite automata. *Inf. Comput.* **205**(8), 1173–1187 (2007). <https://doi.org/10.1016/j.ic.2007.01.008>
15. Kapoutsis, C.: Removing bidirectionality from nondeterministic finite automata. In: Jędrzejowicz, J., Szepietowski, A. (eds.) MFCS 2005. LNCS, vol. 3618, pp. 544–555. Springer, Heidelberg (2005). https://doi.org/10.1007/11549345_47
16. Kapoutsis, C.A.: Two-way automata versus logarithmic space. *Theory Comput. Syst.* **55**(2), 421–447 (2014). <https://doi.org/10.1007/s00224-013-9465-0>
17. Kondacs, A., Watrous, J.: On the power of quantum finite state automata. In: Proceedings of 38th Annual Symposium on Foundations of Computer Science, FOCS 1997, pp. 66–75. IEEE Computer Society (1997). <https://doi.org/10.1109/SFCS.1997.646094>
18. Kunc, M., Okhotin, A.: Describing periodicity in two-way deterministic finite automata using transformation semigroups. In: Mauri, G., Leporati, A. (eds.) DLT 2011. LNCS, vol. 6795, pp. 324–336. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22321-1_28
19. Kunc, M., Okhotin, A.: Reversibility of computations in graph-walking automata. In: Chatterjee, K., Sgall, J. (eds.) MFCS 2013. LNCS, vol. 8087, pp. 595–606. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40313-2_53

20. Landauer, R.: Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.* **5**(3), 183–191 (1961). <https://doi.org/10.1147/rd.53.0183>
21. Lange, K., McKenzie, P., Tapp, A.: Reversible space equals deterministic space. *J. Comput. Syst. Sci.* **60**(2), 354–367 (2000). <https://doi.org/10.1006/jcss.1999.1672>
22. Martynova, O.: Personal Communication, April 2019
23. Morita, K.: A deterministic two-way multi-head finite automaton can be converted into a reversible one with the same number of heads. In: Glück, R., Yokoyama, T. (eds.) *RC 2012. LNCS*, vol. 7581, pp. 29–43. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36315-3_3
24. Muscholl, A., Samuelides, M., Segoufin, L.: Complementing deterministic tree-walking automata. *Inf. Process. Lett.* **99**(1), 33–39 (2006). <https://doi.org/10.1016/j.ipl.2005.09.017>
25. Panaite, P., Pelc, A.: Exploring unknown undirected graphs. *J. Algorithms* **33**(2), 281–295 (1999). <https://doi.org/10.1006/jagm.1999.1043>
26. Rabin, M.O., Scott, D.S.: Finite automata and their decision problems. *IBM J. Res. Dev.* **3**(2), 114–125 (1959). <https://doi.org/10.1147/rd.32.0114>
27. Rollik, H.: Automaten in planaren graphen. *Acta Inform.* **13**, 287–298 (1980). <https://doi.org/10.1007/BF00288647>
28. Sipser, M.: Halting space-bounded computations. *Theoret. Comput. Sci.* **10**, 335–338 (1980). [https://doi.org/10.1016/0304-3975\(80\)90053-5](https://doi.org/10.1016/0304-3975(80)90053-5)
29. Thomas, W.: On logics, tilings, and automata. In: Albert, J.L., Monien, B., Artalejo, M.R. (eds.) *ICALP 1991. LNCS*, vol. 510, pp. 441–454. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-54233-7_154