Available online at www.sciencedirect.com
ScienceDirect
Information and Computation 205 (2007) 1173–1187
Information and Computation
www.elsevier.com/locate/ic

Complementing two-way finite automata☆

Viliam Geffert [a], Carlo Mereghetti [b,*], Giovanni Pighizzini [c]

[a] Department of Computer Science, P.J. Šafárik University, Jesenná 5, 04154 Košice, Slovakia
[b] Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, via Comelico 39, 20135 Milano, Italy
[c] Dipartimento di Informatica e Comunicazione, Università degli Studi di Milano, via Comelico 39, 20135 Milano, Italy

Received 21 July 2005; revised 28 September 2006
Available online 6 April 2007

Abstract

We study the relationship between the sizes of two-way finite automata accepting a language and its complement. In the deterministic case, for a given automaton (2dfa) with n states, we build an automaton accepting the complement with at most $4n$ states, independently of the size of the input alphabet. Actually, we show a stronger result, by presenting an equivalent $4n$-state 2dfa that always halts. For the nondeterministic case, using a variant of inductive counting, we show that the complement of a unary language, accepted by an n-state two-way automaton (2nfa), can be accepted by an $O(n^8)$-state 2nfa. Here we also make 2nfa's halting. This allows the simulation of unary 2nfa's by probabilistic Las Vegas two-way automata with $O(n^8)$ states.

Keywords: Finite state automata; Formal languages; Descriptional complexity

1. Introduction

Automata theory is one of the oldest topics in computer science. In spite of that, there is a renewed interest in this subject recently. In particular, two aspects of automata theory have been extensively investigated: nonstandard models and descriptional complexity.

Nonstandard models of automata (among others, probabilistic [18], Las Vegas [7], self-verifying [2], and quantum [12,16]) differ from classical ones in the evolution rules and/or in the acceptance conditions.

Descriptional complexity compares formal systems with respect to their conciseness. (For a recent survey, see [5].) Several variants of finite automata are known from the literature (one-way or two-way, deterministic or

☆ This work was partially supported by the Science and Technology Assistance Agency under contract APVT-20-004104, the Slovak Grant Agency for Science (VEGA) under contract "Combinatorial Structures and Complexity of Algorithms", and by MIUR under the projects FIRB "Descriptional complexity of automata and related structures" and COFIN "Linguaggi formali e automi: metodi, modelli e applicazioni". A preliminary version of this work was presented at the *9th Int. Conf. Developments in Language Theory, Palermo, Italy, July 4–8, 2005.*

* Corresponding author.
E-mail addresses: viliam.geffert@upjs.sk (V. Geffert), mereghetti@dsi.unimi.it (C. Mereghetti), pighizzi@dico.unimi.it (G. Pighizzini).

0890-5401/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2007.01.008

nondeterministic, ..., see, e.g., [6]). They all have the same computational power. In fact, they characterize the class of regular languages. However, two different models may require a considerably different number of states for the same language. The first widely known result in this sense compares nondeterminism with determinism for one-way finite automata (1nfa versus 1dfa): each n-state 1nfa can be simulated by a 1dfa with 2^n states. Moreover, for each n, there is a language accepted by an n-state 1nfa such that each equivalent 1dfa has at least 2^n states. Thus, in general, we cannot do any better [15,17].

The corresponding exponential gap for two-way machines (2nfa versus 2dfa), conjectured by Sakoda and Sipser in 1978 [19], is still open. In the unary case, i.e., for automata with a single letter input alphabet, a subexponential simulation of 2nfa's by 2dfa's has been obtained [4]. It might be interesting to point out that the 2nfa versus 2dfa question can be regarded as another instance of the ubiquitous nondeterminism versus determinism question, leading to some fundamental problems such as $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ or $\mathrm{L} \stackrel{?}{=} \mathrm{NL}$. Actually, Berman and Lingas show in [1] that if $\mathrm{L} = \mathrm{NL}$ then, for some polynomial p, for all integers m and k-state 2nfa A, there exists a $p(mk)$-state 2dfa accepting the subset of the language $L(A)$, which consists of all strings of lengths not exceeding m. As a consequence of this result, Sipser [20] relates the $\mathrm{L} \stackrel{?}{=} \mathrm{NL}$ question also to the existence of sweeping automata (see Definition 2.1) with a polynomial number of states for a certain family of regular languages. This might give added evidence that the study of descriptional complexity of automata is not only motivated by the investigation on the succinctness of representing regular languages, but is also related to fundamental questions in complexity theory.

In this paper, we study the relationship between the sizes of two-way automata accepting a language and its complement. Related topics for one-way automata are considered in [10,13], while in [23] the construction of 1nfa's accepting the complement of languages accepted by 2nfa's is investigated.

In the deterministic case, for a given 2dfa with n states, we show how to build a 2dfa accepting the complement with at most $4n$ states. This improves the known upper bound [21], from $O(n^2)$ to $O(n)$. The construction does not depend on the size of the input alphabet. In [21], it was pointed out that, besides $O(n^2)$, we can use a modified construction with $O(n \cdot m^2)$ states, where n is the number of states of the original machine and m the size of the input alphabet. This gives a linear upper bound for languages over a *fixed* input alphabet, but not in the general case. For example, the results presented in [10,11,20] consider witness regular languages with the alphabet size growing exponentially in n. Actually, our result is stronger: we prove that each n-state 2dfa can be simulated by a 2dfa with $4n$ states that halts on any input.

For the nondeterministic case, we show that the complement of a unary language, accepted by an n-state 2nfa, can be accepted by an $O(n^8)$-state 2nfa. The construction is based on a variant of the inductive counting [3,9,22]. Here we also prove a stronger result, namely, each unary n-state 2nfa can be replaced by an equivalent $O(n^8)$-state self-verifying automaton (2svfa) which halts on every input. (Self-verifying machines are a special case of nondeterministic machines. The complement for a self-verifying automaton can be immediately obtained by exchanging accepting with rejecting states.)

We were not able to resolve the problem of complement for 2nfa's in the general (nonunary) case. As discussed in Section 6, we notice that stating an exponential gap (in terms of the number of states) between 2nfa's and 2nfa's accepting the complement would imply a positive answer to the conjecture of Sakoda and Sipser. If, moreover, the proof involved only polynomially long strings, this would lead to a separation of L from NL, by the argument of Berman and Lingas, recalled above.

As a consequence of our result concerning the complementation of unary 2nfa's, we also state a connection with Las Vegas automata. In particular, we show that unary 2nfa's can be simulated by two-way Las Vegas automata with a polynomial number of states.

2. Basic definitions

Here, we briefly recall some basic definitions concerning finite state automata. For a detailed exposition, we refer the reader to [6]. Given a set S, $|S|$ denotes its cardinality and 2^S the family of all its subsets. Given an alphabet Σ, the *complement* of a language $L \subseteq \Sigma^*$ is the language $L^c = \Sigma^* \backslash L$.

A *two-way nondeterministic finite automaton* (2nfa, for short) is defined as a quintuple $A = (Q, \Sigma, \delta, q_0, F)$ in which Q is the finite set of states, Σ is the finite input alphabet, $\delta : Q \times (\Sigma \cup \{\vdash, \dashv\}) \to 2^{Q \times \{-1,0,+1\}}$ is the transition

function, $\vdash, \dashv \notin \Sigma$ are two special symbols, called the left and the right endmarker, respectively, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting (final) states. The input is stored on the input tape surrounded by the two endmarkers. The cells of the input tape are numbered from left to right, beginning with zero for the left endmarker. In one move, A reads an input symbol, changes its state, and moves the input head one cell to the right, left, or keeps it stationary, depending on whether δ returns $+1$, -1, or 0, respectively. If, for some $q \in Q$ and $a \in \Sigma \cup \{\vdash, \dashv\}$, we have $|\delta(q,a)| > 1$, the machine makes *a nondeterministic choice*. If $|\delta(q,a)| = 0$, the machine *halts*.

The machine accepts the input, if there exists a computation path starting from the initial state q_0 with head on the left endmarker and reaching an accepting state $q \in F$. The language accepted by A, denoted by $L(A)$, consists of all input strings that are accepted.

An automaton is a two-way *deterministic* finite state automaton (2dfa), whenever $|\delta(q,a)| \leqslant 1$ for each $q \in Q$ and $a \in \Sigma \cup \{\vdash, \dashv\}$. With a slight abuse of notation, we shall then write $\delta(q,a) =$ undefined, instead of $\delta(q,a) = \emptyset$, and $\delta(q,a) = (q',d)$, instead of $\delta(q,a) = \{(q',d)\}$.

A (non)deterministic automaton is *one-way* (1nfa or 1dfa), if it never moves the head to the left, i.e., if $(q',d) \in \delta(q,a)$ implies $d \neq -1$.

We call *unary* any automaton that works with a single letter input alphabet. An automaton is *halting* if no computation path can get into an infinite loop, that is, on every input, each computation path halts after a finite number of steps.

We say that an automaton A is *almost equivalent* to an automaton A', if the languages accepted by A and A' coincide, with the exception of a finite number of strings. If these two languages coincide on all strings, with no exceptions, A and A' are (*fully*) *equivalent*.

In what follows, we will be particularly interested in weaker versions of 2nfa's and 2dfa's:

Definition 2.1.

- A *quasi-sweeping* [1] automaton (qsnfa) is a 2nfa performing *both* input head reversals and nondeterministic choices only at the endmarkers [14]. If, moreover, the above automaton is deterministic, we call it *sweeping* (qsdfa) [20].
- A two-way *self-verifying* automaton A (2svfa) [2] is a 2nfa which, besides the set of accepting states $F \subseteq Q$, is equipped also with $F^r \subseteq Q$, a set of so-called rejecting states. For each input $w \in L(A)$, there exists at least one computation path halting in an accepting state $q \in F$, and no path may halt in a state $q \in F^r$. Conversely, for $w \notin L(A)$, there exists at least one path halting in a rejecting $q \in F^r$, and no path halts in a state $q \in F$. A 2svfa can be quasi-sweeping (qssvfa) or one-way (1svfa), with the obvious meaning.
- A two-way *Las Vegas* finite state automaton A (2lvfa) [7] is a 2svfa equipped with probabilistic transitions. If, for some $q \in Q$ and $a \in \Sigma \cup \{\vdash, \dashv\}$, we have $\delta(q,a) = \{(q_1,d_1), \ldots, (q_h,d_h)\}$, then the transition (q_i,d_i) is chosen with the probability of $1/h$. To make these values more flexible, we allow repetitions in the list $(q_1,d_1), \ldots, (q_h,d_h)$.
  If $w \in L(A)$, then A halts in some accepting state $q \in F$ with a probability of at least $1/2$, but the probability of halting in a rejecting state $q \in F^r$ is zero. Conversely, for $w \notin L(A)$, the probability of halting in some rejecting state $q \in F^r$ is at least $1/2$, but it is zero for $q \in F$.

Note that some computation paths of a 2svfa may result in a "don't-know" answer, since the machine may also halt in states that are neither in F nor in F^r, or it may get into an infinite loop. Self-verifying two-way automata stand in between the ordinary 2nfa's and 2dfa's. For example, a machine for the complement of $L(A)$ can be immediately obtained from A by exchanging accepting with rejecting states.

A Las Vegas automaton, as a special case of the self-verifying automaton, never returns a wrong answer. (This distinguishes Las Vegas from other probabilistic models, that can reply incorrectly, though with a small probability.) Moreover, the probability of a "don't-know" answer is below $1/2$. (This value can be reduced to

[1] The term sweeping is introduced in [20], to denote automata performing input head reversals only at the endmarkers, with no nondeterministic choices at all. Here, our designation "quasi-sweeping" denotes a relaxation of this paradigm, where both the head reversals and nondeterministic choices can be taken at the endmarkers only.

$1/2^k$, by restarting the computation k times from the very beginning.) Thus, Las Vegas algorithms are facing reality squarely, they are trustworthy and easy to implement on realistic computers.

3. Complement for deterministic machines

In this section, we show that, for deterministic two-way finite state automata, construction of an automaton for the complement of the language accepted by the original machine requires only a linear increase in the number of states. More precisely, we show that any n-state 2dfa A can be transformed into a $4n$-state 2dfa A' that halts on every input, and accepts the complement of the language accepted by A. Moreover, if the original machine already halts on every input, then n states are sufficient, that is, converting A into A' does not increase the number of states. As a consequence, we also get that any n-state 2dfa A can be replaced by an equivalent $4n$-state 2dfa A' that always halts. This reduces the known upper bound [21], from $O(n^2)$ to $O(n)$.

We start by putting the given 2dfa A into some kind of normal form, which keeps the technicalities of our main construction simple. (This model is a natural counterpart of the usual two-way Turing machine and its acceptance conditions. The minor differences between the model used here and the one used in classical textbooks/papers will be discussed later, at the end of this section.)

First, we can assume that, if the machine A accepts the input, it halts in a unique accepting state q_f, with the input head parked at the left endmarker. This can be achieved by choosing one of the accepting states in F as q_f. Then, for each $q \in F \setminus \{q_f\}$, we can redefine $\delta(q,a) = (q_f, 0)$, for each $a \in \Sigma$. Finally, to make A halt at the left endmarker, we redefine $\delta(q_f, a) = (q_f, -1)$, for each $a \neq \vdash$, but $\delta(q_f, \vdash) =$ undefined.

The above modification does not work for $F = \emptyset$, since then we cannot pick up a state q_f in F. However, this can happen only if A accepts the empty language. But both $\emptyset$ and Σ^* can be accepted by a 2dfa having a single state, namely, the initial state q_0, with $q_0 \notin F$ (or, respectively, $q_0 \in F$), and with $\delta(q_0, \vdash) =$ undefined, that is, halting at the very beginning, without executing a single computation step. In what follows, we shall therefore assume that $L(A)$ does not coincide with $\emptyset$ or Σ^*, and hence A has a unique accepting state q_f, different from the initial state q_0.

Finally, we can also assume that A does not perform stationary moves. Suppose that, in a state q with the input head scanning a symbol a, the automaton performs a stationary move, i.e., $\delta(q,a) = (q', 0)$, for some $q' \in Q$. Two possibilities arise: either (i) after a sequence of stationary moves, A will finally move the input head to the left or right, or (ii) A will never change the input head position, that is, A will loop forever or will halt the computation on the same cell. Observe that the choice between (i) and (ii) depends only on the current state q and the input symbol a, and hence all details can be determined by inspecting the transition table for the function δ. This enables us to modify the automaton A as follows, so as to avoid stationary moves:

- In case (i), we "shortcut" the value of $\delta(q,a)$ by redefining it as $\delta(q,a) = (p,d)$, where $p \in Q$ and $d \in \{-1,+1\}$ correspond to the first move of the head to a different cell.
- In case (ii), we consider two subcases. If $a \neq \vdash$ then, by our previous assumption about the final state q_f, we can conclude that the computation is not accepting. Hence, we just make $\delta(q,a)$ undefined. On the other hand, if $a = \vdash$, the computation could still be accepting. By inspecting the transition table, starting from the state q with the input head parked on the left endmarker, we decide whether the final state q_f is reachable by the subsequent stationary moves. If this is the case, we set $\delta(q, \vdash) = (q_f, +1)$, otherwise we make $\delta(q, \vdash)$ undefined.

All this reasoning enables us to make the following assumptions on the given 2dfa A, *without increasing* its number of states:

Lemma 3.1. *Each n-state 2dfa can be replaced by an equivalent 2dfa A, with at most n states, such that*

- *A has exactly one accepting state q_f, different from the initial state q_0, and halts on every accepted input on the left endmarker. In addition*:

$$\delta(q_f, a) = \begin{cases} (q_f, -1) & \text{if } a \neq \vdash, \\ \text{undefined} & \text{if } a = \vdash. \end{cases}$$

- *A does not perform stationary moves.*
- *If, moreover, the original machine halts on every input, then so does A.*

Now we can turn our attention to the problem of replacing the given automaton A by a machine A' accepting the complement of $L(A)$. Since A is deterministic, such a construction causes no problems, *provided that A halts on every input*:

First, put the machine A into the normal form presented in Lemma 3.1. Then δ', the transition function for the automaton A', is obtained as follows:

- If, for some $q \in Q\backslash\{q_f\}, a \in \Sigma \cup \{\vdash, \dashv\}$, and $d \in \{-1, +1\}$, we have $\delta(q, a) = (q_f, d)$, then let $\delta'(q, a) =$ undefined.
- Similarly, if $\delta(q, a) =$ undefined, then $\delta'(q, a) = (q_f, -1)$ for $a \neq \vdash$, but $\delta'(q, a) = (q_f, +1)$ for $a = \vdash$.
- Otherwise, $\delta'(q, a) = \delta(q, a)$. This includes the case of $q = q_f$, that is, $\delta'(q_f, a) = \delta(q_f, a)$, as presented in Lemma 3.1.
- The initial and final states of A' are the same as those of A.

Since, by assumption, A halts on every input and, by Lemma 3.1, $q_0 \neq q_f$, it is obvious that A' accepts if and only if A does not accept.

Corollary 3.2. *For each n-state 2dfa A that halts on every input, there exists an n-state 2dfa A' accepting the complement of $L(A)$, also halting on every input.*

Note that the assumption about halting is essential for the trivial construction above. If, for some input $w \in \Sigma^*$, A gets into an infinite loop, then A' will get into an infinite loop as well, and hence w will be rejected both by A and A'.

To avoid this problem, we shall now present the construction of the halting 2dfa A', by suitably refining Sipser's construction for space bounded Turing machines [21]. To make our result more readable, we first recall the original construction of Sipser (restricted to the case of 2dfa in the normal form presented in Lemma 3.1).

For each $w \in \Sigma^*$, a deterministic machine accepts w if and only if there is a "backward" path, following the history of computation in reverse, from the unique accepting configuration $(q_f, 0)$ to the unique initial configuration $(q_0, 0)$. A "configuration" is a pair (q, i), where $q \in Q$ is the current finite control state and $i \in \{0, \ldots, |w| + 1\}$ the current position of the input head.

Consider the graph whose nodes represent configurations and edges computation steps. Since A is deterministic, the component of the graph containing $(q_f, 0)$ is a tree rooted at this configuration, with backward paths branching to all possible predecessors of $(q_f, 0)$. In addition, no backward path starting from $(q_f, 0)$ can cycle (hence, it is of finite length), because the halting configuration $(q_f, 0)$ cannot be reached by a forward path from a cycle.

Thus, the machine for the complement of $L(A)$ can perform a depth-first search of this tree in order to detect whether the initial configuration $(q_0, 0)$ belongs to the predecessors of $(q_f, 0)$. If this is the case, the simulator rejects. On the other hand, if the whole tree has been examined without reaching $(q_0, 0)$, the simulator accepts.

The depth-first search strategy visits the first (in some fixed lexicographic order) immediate predecessor of the current configuration that has not been visited yet. If there are no such predecessors, the machine travels along the edge toward the unique immediate successor. (Traveling forward along an edge is simulated by executing a single computation step of A, traveling backward is a corresponding "undo" operation.)

For this search, the simulator has only to keep, in its finite control, the state q related to the currently visited configuration (q, i) (the input head position i is represented by its own input head), together with the information about the previously visited configuration (its state and input head position relative to the current head position, i.e., a number ± 1). Hence, the simulator uses $O(n^2)$ states.

Now, we present our improvements on this procedure. First, fix a linear order on the state set of the original automaton. As usual, the symbols "$<$" and "$>$" denote the ordering relation.

Our implementation of the depth-first search examines each configuration (q,i) in two modes: (1) examination of the "left" predecessors of (q,i), that is, immediate predecessors with input head at the position $i-1$, (2) examination of the "right" predecessors, with head position $i+1$. For each $q \in Q$ and each mode, we introduce a starting and a finishing state. So the machine for the complement will use the following set of states:

$$Q' = \{q\nwarrow,\ q\downarrow_1,\ q\nearrow,\ q\downarrow_2 : q \in Q\}\,.$$

These $4n$ states are interpreted as follows:

- $q\nwarrow$ Starting state for Mode 1, examination of left predecessors for the configuration (q,i). A left predecessor is a configuration $(p,i-1)$, with the input head scanning a symbol a, such that $\delta(p,a) = (q,+1)$. Left predecessors will be examined one after another, according to the linear order induced by the relation "$<$". To inspect the content of the input square $i-1$ (that is, the symbol a), the simulator A' (if it is in the state $q\nwarrow$) has its input head one position to the left of the actual position of the original machine A in configuration (q,i).
- $q\downarrow_1$ Finishing state for Mode 1. All the left predecessors of (q,i) have been examined, but we still have to examine the right predecessors of (q,i). In the state $q\downarrow_1$, the input head of the simulator A' is in the actual position, i.e., the position i.
- $q\nearrow$ Starting state for Mode 2, examination of right predecessors for (q,i), when the left predecessors have been finished. A right predecessor is a configuration $(p,i+1)$, with the head scanning a symbol a, such that $\delta(p,a) = (q,-1)$. The right predecessors will also be examined in the linear order induced by "$<$". In the state $q\nearrow$, the simulator A' has its input head one position to the right of the actual position of the configuration (q,i), to inspect the symbol a in the input square $i+1$.
- $q\downarrow_2$ Finishing state for Mode 2. Both the left and the right predecessors of (q,i) have been examined. In the state $q\downarrow_2$, the input head of A' is in the actual position, i.e., the position i.

Let us now describe the transition function $\delta' : Q' \times (\Sigma \cup \{\vdash, \dashv\}) \to Q' \times \{-1,0,+1\}$ implementing this strategy. For each (type of) state $q' \in Q'$ and each symbol $a \in \Sigma \cup \{\vdash, \dashv\}$, we first display a procedure that assigns a value of $\delta'(q',a) \in Q' \times \{-1,0,+1\}$ to the transition table and, after that, we present an explanation for this procedure. Note that A' will use stationary moves. The reader should also keep in mind that the procedures displayed below are *not* executed by the machine A' but, rather, they are used to fill in the entries in the transition table for A'.

Transition $\delta'(q\nwarrow, a)$*:*

```
11: if   a = ⊣ then   δ'(q↖, a) := undefined
12: else if   there is no p ∈ Q : δ(p, a) = (q, +1) then   δ'(q↖, a) := (q↓1, +1)
13:      else let   q̃ := min{p ∈ Q : δ(p, a) = (q, +1)} ;
14:           if   a ≠ ⊢ then   δ'(q↖, a) := (q̃↖, −1)
15:           else if   q̃ = q0 then   δ'(q↖, a) := undefined
16:                else   δ'(q↖, a) := (q̃↓1, 0)
17: end   end   end   end
```

Recall that A' gets to the state $q\nwarrow$ when, for some i, it starts the examination of the left predecessors of the configuration (q,i), that is, of configurations $(p,i-1)$ such that $\delta(p,a) = (q,+1)$, where a is the content of the cell $i-1$. By definition of $q\nwarrow$, A' has its input head already at the position $i-1$. (This implies that A' can never reach $q\nwarrow$ with $a = \dashv$. Line 11 is given just for completeness, to fill in all entries in the transition table for δ'.)

In line 13, we select the first left predecessor in the lexicographic order, i.e., a configuration $(\tilde{q}, i-1)$, and start to examine this configuration with the same method. To this aim, we switch the state to $\tilde{q}\nwarrow$, and move the head one position to the left of $i-1$ (line 14). There are two special cases:

First, (q,i) may have no left predecessors, i.e., the set $\{p \in Q : \delta(p,a) = (q,+1)\}$ is empty (line 12). Thus, we can terminate Mode 1 for the current configuration, as if all left predecessors had been searched, by switching to $q\downarrow_1$ and moving the head to the position i.

Second, $(\tilde{q}, i-1)$ is the first left predecessor, but the input head is on the left endmarker, i.e., $a = \vdash$ and $i-1 = 0$. This means that the configuration we have to examine, i.e., $(\tilde{q}, 0)$, does not have left predecessors. If $\tilde{q} = q_0$, then the initial configuration $(q_0, 0)$ is in the tree, meaning that A accepts. Hence, we immediately let A' reject the input by setting $\delta'(q\nwarrow, a) :=$ undefined at line 15. If $\tilde{q} \neq q_0$, we start the examination of $(\tilde{q}, 0)$ by directly finishing Mode 1, in the state $\tilde{q}\downarrow_1$ (line 16).

Transition $\delta'(q\downarrow_1, a)$*:*

21: **if** $a \neq \dashv$ **then** $\delta'(q\downarrow_1, a) := (q\nearrow, +1)$
22: **else** $\delta'(q\downarrow_1, a) := (q\downarrow_2, 0)$
23: **end**

In this state, the examination of the left predecessors of (q, i) has been completed. Hence, the search continues with the examination of the right predecessors in Mode 2 (line 21), by switching to the state $q\nearrow$ and moving the head to the position $i+1$.

If the input head is already on the right endmarker, i.e., $a = \dashv$, then the configuration (q, i) does not have any right predecessors (line 22). Hence, by switching to $q\downarrow_2$, we finish Mode 2 immediately, as if all right predecessors had been searched.

Transition $\delta'(q\nearrow, a)$*:*

31: **if** $a = \vdash$ **then** $\delta'(q\nearrow, a) :=$ undefined
32: **else if** there is no $p \in Q : \delta(p, a) = (q, -1)$ **then** $\delta'(q\nearrow, a) := (q\downarrow_2, -1)$
33: **else let** $\tilde{q} := \min\{p \in Q : \delta(p, a) = (q, -1)\}$;
34: $\delta'(q\nearrow, a) := (\tilde{q}\nwarrow, -1)$
35: **end end**

In this state, A' starts to examine right predecessors of (q, i), i.e., configurations $(p, i+1)$ such that $\delta(p, a) = (q, -1)$, where a is the content of the input square $i+1$. A' has its head already at the position $i+1$. (This implies that A' can never reach $q\nearrow$ with $a = \vdash$. Line 31 is given just for completeness, cf. line 11 for $q\nwarrow$.)

In line 33, we select $(\tilde{q}, i+1)$, the first right predecessor of (q, i), and start to examine it with the same method. (Among others, the left predecessors of $(\tilde{q}, i+1)$ are going to be examined.) To this aim, we switch to $\tilde{q}\nwarrow$, and move the head one position to the left of $i+1$ (line 34).

There is only one special case here, when there are no right predecessors of (q, i), i.e., the set $\{p \in Q : \delta(p, a) = (q, -1)\}$ is empty (line 32). We can finish Mode 2 immediately, which completes the search for (q, i).

Transition $\delta'(q\downarrow_2, a)$*:*

41: **if** $q = q_0$ and $a = \vdash$, or $q = q_f$ and $a = \vdash$, or $\delta(q, a) =$ undefined **then**
42: $\delta'(q\downarrow_2, a) :=$ undefined
43: **else let** $(r, d) := \delta(q, a)$;
44: **if** there is no $p \in Q : p > q$ and $\delta(p, a) = (r, d)$ **then**
45: **if** $d = +1$ **then** $\delta'(q\downarrow_2, a) := (r\downarrow_1, +1)$
46: **else** $\delta'(q\downarrow_2, a) := (r\downarrow_2, -1)$
47: **end**
48: **else let** $\tilde{q} := \min\{p \in Q : p > q$ and $\delta(p, a) = (r, d)\}$;
49: **if** $a \neq \vdash$ **then** $\delta'(q\downarrow_2, a) := (\tilde{q}\nwarrow, -1)$
50: **else** $\delta'(q\downarrow_2, a) := (\tilde{q}\downarrow_1, 0)$
51: **end end end**

This state concludes the examination of the configuration (q, i), and all configurations in the subtree rooted in (q, i). The machine A' has its head at the position i. We can recover the immediate successor of (q, i), as follows: let $\delta(q, a) = (r, d)$, for some $r \in Q$ and $d \in \{-1, +1\}$ (line 43). Then the successor of (q, i) is (r, j), with $j = i+d$.

With the value of (r, j) in our hands, we can decide (line 44), whether (q, i) is the "last" left/right predecessor of (r, j). If this is the case, and $d = +1$ (line 45), we know that (q, i) is the last left predecessor, hence, Mode 1 is over for (r, j). We thus switch to the state $r\downarrow_1$, and move the head in the correct direction. Similarly, if $d = -1$ (line 46), (q, i) is the last right predecessor, which requires to terminate Mode 2 for (r, j), by switching to $r\downarrow_2$.

Conversely, if (q, i) is not the last left/right predecessor of (r, j), we find $(\tilde{q}, i)$, the next larger predecessor of the same kind (line 48), and start to examine it with the same method (line 49). If $a = \vdash$ (line 50), we skip Mode 1 for $(\tilde{q}, i)$, since it has no left predecessors.

There are three special cases, described in lines 41–42. First, if $q = q_0$ and $a = \vdash$, the initial configuration $(q_0, 0)$ has been reached by a backward path starting from $(q_f, 0)$. Thus, the machine A', accepting the complement of $L(A)$, stops in the rejecting state $q_0\downarrow_2$ (line 42).

Second, if $q = q_f$ and $a = \vdash$, the whole tree rooted in $(q_f, 0)$ has been explored without retrieving the initial configuration $(q_0, 0)$. Thus, A rejects and hence A' accepts by remaining in the final state $q_f\downarrow_2$ (line 42).

Third, the procedure handling the "standard" case (lines 43–51) does not work properly, if $\delta(q, a) =$ undefined. However, A' visits only configurations from which A has a forward path to the halting configuration $(q_f, 0)$. This implies that A' can never visit a configuration (q, i) such that $q \neq q_f$ and $\delta(q, a) =$ undefined. For completeness, to fill in entries for all combinations of $q \in Q$ and $a \in \Sigma \cup \{\vdash, \dashv\}$ in the transition table for δ', we just make such values undefined (line 42).

Initial and final states:

$$q'_0 := q_f\downarrow_1; \quad q'_f := q_f\downarrow_2$$

Recall that the original machine A accepts by halting in $(q_f, 0)$, and this configuration does not have any left predecessors. Thus, to decide whether an input is accepted, we can start the depth-first search from the state $q_f\downarrow_1$ with the head on the left endmarker. By the previous discussion, we see that (i) either the input is accepted by A, and then the machine A' aborts its search in the state $q_0\downarrow_2$ at the left endmarker, (ii) or the input is rejected, and then A' stops by reaching $q_f\downarrow_2$ at the same endmarker. (Both cases are related to undefined transitions in line 42.)

This construction leads to

Theorem 3.3. *For each n-state 2dfa A, there exists a* $4n$*-state 2dfa* A' *accepting the complement of* $L(A)$. *Moreover,* A' *halts on every input.*

By applying Lemma 3.1 to the automaton A' constructed in Theorem 3.3, we can remove stationary moves and put it into the normal form. Then, by using Corollary 3.2, i.e., by making a complement of the complement, we have:

Corollary 3.4. *Each n-state 2dfa A can be replaced by an equivalent* $4n$*-state 2dfa* A' *that halts on every input.*

It is easy to modify the constructions given above for the classical model of 2dfa (see, e.g., [19]), which differs in the following:

The machine A accepts the input if it gets to an accepting state $q \in F$ (i.e., A can potentially use more than one accepting state), and, *at the same time*, A halts in q with the input head at the right endmarker.

For this computational model, the simulation presented above should be modified as follows:

First, replace the original machine A by an equivalent 2dfa A' having a single accepting state q_f, not increasing the number of states. This can be done in the same way as already described above, that is, fix any state in F as q_f, and replace halting in any other state $q \in F\backslash\{q_f\}$ by halting in q_f, that is, we let $\delta'(q, \dashv) = (q_f, 0)$. After that, modify the transitions of A' so that it does not perform any stationary moves. (For details, see discussion above, preceding Lemma 3.1.)

Second, replace A' by A'' accepting the complement of the original language and using $4n+1$ states. A'' uses a new initial state q''_0 in which it scans the input to the right endmarker, where it switches to $q_f\nwarrow$ with the input head one position to the left of the right endmarker. That is, $\delta''(q''_0, a) = (q''_0, +1)$, for each $a \neq \dashv$, but $\delta''(q''_0, \dashv) = (q_f\nwarrow, -1)$. This ensures that the backward search starts from the root of the search tree, i.e., from the unique final configuration of the modified computational model. The rest of the simulation is the same as described earlier, with the following modification in line 41:

41: **if** $q = q_0$ and $a = \vdash$, or $q = q_f$ and $a = \dashv$, or $\delta(q, a) =$ undefined **then**
42: $\quad \delta'(q\downarrow_2, a) :=$ undefined
⋮ ⋮

This ensures that, if $q = q_f$ and $a = \dashv$, i.e., if the machine gets back to the unique accepting and halting configuration, and hence the whole tree has been explored without retrieving the initial configuration, A'' accepts by remaining in the final state $q_f\downarrow_2$.

Theorem 3.5. *For each n-state 2dfa A accepting at the right endmarker, there exists a* $(4n+1)$*-state 2dfa* A'' *for the complement of* $L(A)$*, accepting again at the right endmarker. Moreover,* A'' *halts on every input.*

4. Complement for unary nondeterministic machines

This section is devoted to the problem of complement for nondeterministic two-way automata. As we will discuss in Section 6, for arbitrary alphabets this problem is related to the most famous open question in this field, posed by Sakoda and Sipser in 1978 [19].

However, the situation is different for the case of unary regular languages. Using a modified version of inductive counting [3,9,22], we first replace a given unary 2nfa by an equivalent two-way automaton that is quasi-sweeping, self-verifying, and halting, using only a polynomial number of states. For such an automaton, building an automaton for the complement of the language is straightforward.

First, we need to make our 2nfa quasi-sweeping. The following theorem states that, for sufficiently large unary inputs, nondeterminism as well as input head reversals can be restricted to the moments when the input head scans either of the endmarkers. This costs only a linear increase in the number of states:

Theorem 4.1 (Geffert, Mereghetti, and Pighizzini [4, Thm. 2])**.** *For each n-state unary 2nfa, there exists an almost equivalent qsnfa A such that*:

- A has no more than $2n+2$ states.
- The language $L(A)$ coincides with the original language on strings of length greater than $5n^2$.
- For each accepted input, A has at least one computation path halting with the head parked at the left endmarker, in a unique accepting state q_f.
- The machine A does not perform any stationary moves, with the exception of the last computation step, when it enters q_f.

Besides restricting nondeterminism and reversals, the construction behind Theorem 4.1 removes stationary moves, and makes the machine accept at the left endmarker. For the technical implementation of these marginal details, similar to those presented in Lemma 3.1, the reader is referred also to Lemmas 1 and 2 in [4]. However, unlike in Lemma 3.1, the state q_f in Theorem 4.1 is a *new* state, reached by stationary moves.

Theorem 4.1 allows us to consider a qsnfa A almost equivalent with the original 2nfa. Hence, any accepting computation follows a very regular pattern. Starting from the initial state q_0 at the left endmarker, it makes a nondeterministic choice and moves the input head one position to the right. From this point forward, the input is scanned deterministically from left to right, until the head gets to the right endmarker. This piece of deterministic computation is called a *left-to-right traversal.* At the right endmarker, the machine makes a nondeterministic choice again, moving the head one position to the left. After that, a *right-to-left traversal* deterministically scans the input.

This alternation of nondeterministic choices at the endmarkers and deterministic input traversals is repeated, until the final state q_f is reached, by a single stationary move at the left endmarker.

In our simulation, we shall need a linear order on the set Q, as in Section 3. Moreover, it will be useful to introduce a set $Q_f \subseteq Q$ of states possibly leading to acceptance, that is, the set from which the final state q_f is reachable by a single stationary move on the left endmarker:

$$Q_f = \{q \in Q : (q_f, 0) \in \delta(q, \vdash)\}\,.$$

Now we are ready to present the main result of this section, the simulation of A by an automaton that is self-verifying.

For the reader's ease of understanding, we prefer to present the simulating qssvfa A' in the form of an algorithm, written as high-level code. We will then informally discuss the actual implementation, evaluating the number of states required. In the following code, we use two subroutines whose implementation will be detailed later:

- *simulation*(t): a nondeterministic function, returning a nondeterministically chosen state q that is reachable by a computation path of A in exactly t traversals from the initial configuration. The call of this function may also abort the entire simulation by halting in a "don't-know" state $q_?$, due to a wrong sequence of nondeterministic guesses, if the chosen path halts too early, not having completed t traversals.
- *reach*($q_{\rm prev}, q', {\rm dir}$): a deterministic function, with dir $\in \{0,1\}$. It returns *true*/*false*, depending on whether the state q' can be reached from the state $q_{\rm prev}$ by a left-to-right traversal of the input (for dir $=0$), or by a right-to-left traversal (for dir $=1$).

The nondeterministic simulation algorithm, based on the well-known inductive counting technique, is displayed in Fig. 1.

Basically, the algorithm proceeds by counting, for $t = 0, \ldots, 2|Q| - 1$, where Q is the state set of A, the number of states reachable by A at the endmarkers by all computation paths starting from the initial configuration and traversing the input exactly $t+1$ times. As a side effect of this counting, the algorithm generates all states reachable at the endmarkers, and hence it can correctly decide whether to accept or reject the given input.

Recall that, for each accepted input, A has at least one accepting path that halts at the left endmarker. A simple counting argument shows that then the input must also be accepted by a path with no more than $2|Q| - 1$ traversals. Otherwise, the machine A would repeat the same state on the same endmarker. We can remove such a computational loop from the path, which gives an accepting computation with a smaller number of traversals. For this reason, the loop running for $t = 0, \ldots, 2|Q| - 1$ (nested between lines 2 and 14) suffices to detect an accepting computation.

At the beginning of the tth iteration of this loop, a variable m' contains the exact number of states reachable at the endmarkers by all computation paths that traverse the input exactly t times. (Initially, in line 1, we prepare $m' = 1$ for $t = 0$, the only state reachable by no traversals being the initial state q_0.) In line 3, we save the "old"

```
 1:      m' := 1;
 2:      for t := 0 to 2|Q| − 1 do
 3:          m := m'; m' := 0;
 4:          foreach q' ∈ Q do
 5:              for i := 1 to m do
 6:                  q := simulation(t);
 7:                  if i > 1 and q ≤ q_prev then halt in q_?;
 8:                  q_prev := q;
 9:                  if reach(q_prev, q', t mod 2) then
10:                      if t is odd and q' ∈ Q_f then halt in q_yes;
11:                      m' := m'+1; goto next_q'
12:              end end
13: next_q':  end
14:      end;
15:      halt in q_no
```

Fig. 1. The simulation procedure.

value of m' in the variable m, and clear m' for counting the number of states reachable upon completing one more traversal (i.e., with exactly $t+1$ traversals).

The value of m' is computed in the loop nested between lines 4 and 13, running for each state $q' \in Q$. For each q', we test whether it is reachable by a path with exactly $t+1$ traversals. If it is, we increment the value of m'.

Let us now explain in detail, how to decide whether a given state q' can be reached by exactly $t+1$ traversals. This only requires to generate all m states that are reachable at the endmarkers by all computation paths traversing the input exactly t times, and verify if q' can be reached from any of these by a single traversal. These m states are generated one after another, in the variable q, by using the subroutine *simulation*(t) in line 6, in the loop nested between lines 5 and 12, running for $i = 1, \ldots, m$.

For each generated state q, we test whether q' is reachable from q by a single traversal along the input, by calling the subroutine *reach*[2] in line 9. A small but important detail is that we check left-to-right traversals, if t is even, but right-to-left traversals, for t odd (third parameter of *reach*). In case such a test has a positive outcome for some q, we increment the value of m' by one in line 11, and skip immediately to another state $q' \in Q$, by the goto-statement. This avoids testing the states q not examined yet (the iteration of the loop in lines 5–12 is aborted), which ensures that, even if q' is reachable from several different states q, the value of m' is incremented only once.

Note that the body of the if-statement, nested between lines 9 and 12, is executed for each $t = 0, \ldots, 2|Q| - 1$ and for each $q' \in Q$ that is reachable by any path with exactly $t+1$ traversals. That is, it is executed for each q' reachable at the endmarkers. This allows us to test if the machine A can get into its final state q_f. Since this happens by a stationary move at the left endmarker, we check whether $q' \in Q_f$ for odd values of t, in line 10. If this is the case, we abort the entire computation by halting in the state q_{yes}, the input is accepted.

On the other hand, if the iteration of the outermost loop has been completed for each $t = 0, \ldots, 2|Q| - 1$, no reachable $q' \in Q_f$ has been found at the left endmarker. (Otherwise, the search would have stopped already, in line 10.) This implies that the input is not accepted. Therefore, in line 15, we stop in the rejecting state q_{no}.

However, we have to ensure that the loop running for $i = 1, \ldots, m$, nested between lines 5 and 12, will correctly generate all m states that are reachable at the endmarkers by computation paths traversing the input exactly t times. This is done by calling the subroutine *simulation*(t) in line 6, which generates, in the variable q, one state reached nondeterministically after t traversals. This does not imply that, in the course of m iterations of the loop, the subroutine will generate such states without repetitions, which would produce all of them. However, for the right sequence of nondeterministic guesses, the m iterations of $q :=$ *simulation*(t) will produce, one after another, all m states without repetitions, moreover, according to the fixed linear order on the state set Q.

This can be verified without explicitly storing all generated states; we only have to keep, in a single variable q_{prev}, the state q resulting from the previous iteration (line 8), and to check (line 7) whether the "new" generated value of q is strictly larger than that generated in the previous iteration. If the generated sequence of states is not strictly increasing, the computation is aborted by halting in the "don't-know" state $q_?$. For the right sequence of nondeterministic guesses, generating always the strictly increasing sequences of states, the condition of the if-test in line 7 is not satisfied during the entire computation. If this happens, we can be sure that all values and results have been computed correctly.

It is not hard to see that: (i) if the input is accepted by A, at least one computation path halts in the state q_{yes}, and no path halts in q_{no}, (ii) if the input is rejected, at least one path halts in q_{no}, and no path halts in q_{yes}. (iii) Due to wrong sequences of nondeterministic guesses, some computation paths halt in $q_?$ (don't-know), but no path can get into an infinite loop.

Let us now present the subroutine *simulation*(t). Again, it will be displayed as a high-level code, informally suggesting its implementation on a finite state machine. It returns a nondeterministically chosen state q, reachable by A in exactly t traversals from the initial configuration.

function *simulation*(t): state;

[2] The use of q_{prev} as a first parameter, instead of q, will do no harm, since q_{prev} contains a copy of q, by line 8. This trick will allow us to modify the value of q in the implementation of *reach* (see below), saving in this way one variable.

```
6.1:      q := q0; t̃ := t;
6.2:      move the head to the left endmarker;
6.3:      while t̃ > 0 do
6.4:           keeping the current state in q,
                    perform a direct nondeterministic simulation of A,
                    until the head hits the opposite endmarker
                    (if the chosen computation path halts before it hits
                    the opposite endmarker, halt in q?);
6.5:           t̃ := t̃−1
6.6: end  end
```

To implement this routine, we introduce the variable $\tilde{t}$ counting the number of traversals still to be performed. To minimize the number of used variables, the global variable q from the main procedure (line 6 in Fig. 1) is directly employed for the simulation. The variable q stores the current state during the simulation of A, and therefore q also ends up containing the state in which such a simulation possibly ends. If t traversals cannot be completed along the chosen computation path, the routine halts in the don't-know state $q_?$, which aborts the main program.

Let us now show $reach(q_{\text{prev}}, q', \text{dir})$. It yields *true*/*false*, depending on whether the state q' can be reached from the state q_{prev} by a left-to-right traversal of the input (for dir = 0), or by a right-to-left traversal (for dir = 1).

```
     function  reach(q_prev, q', dir): boolean;
9.1:      if  dir = 0 then        – traversals from left to right
9.2:           foreach  q such that (q, +1) ∈ δ(q_prev, ⊢) do
9.3:                q̃ := q;
9.4:                move the head one position to the right of the left endmarker;
9.5:                keeping the current state in q̃,
                         perform the direct deterministic simulation of A,
                         until the head hits the opposite endmarker;
9.6:                if  q̃ = q' then return  true
9.7:           end ;
9.8:           return  false
9.9:      else                    – that is, dir = 1, traversals from right to left
  ⋮            ⋮                  – implemented symmetrically
9.17: end  end
```

We shall describe how to detect a left-to-right traversal (the opposite direction is implemented symmetrically). To minimize the number of used variables, we use the global variable q from the main procedure (Fig. 1) as a control variable for a cycle iterated inside, in lines 9.2–9.7. (Note that the call of the subroutine *reach* is activated in the main procedure below line 8, when the value stored in q can be destroyed.)

This cycle is iterated over all states q arising from q_{prev} at the left endmarker by a nondeterministic step moving the input head to the right. For each such q, an input traversal is simulated, using the local variable $\tilde{q}$. The subroutine returns *true*, if the target state q' is reached by at least one traversal, *false* otherwise. Note that, since A is quasi-sweeping, the simulation of a traversal is deterministic after the first step at the left endmarker.

Summing up, it is easy to see that the simulation algorithm in Fig. 1 performs nondeterministic choices and input head reversals at the endmarkers only, during the calls of *simulation* and *reach*. To see that it can be implemented on a finite state automaton, let us now quickly examine the "amount of information" required during the computation, by considering the variables involved. Recall that the qsnfa A, constructed in Theorem 4.1, has $n' \leqslant 2n+2$ states.

- m' and m are counters for the number of states reached either at the left or at the right endmarker. Hence, their maximum value is n'.
- t is the control variable for the loop between lines 2 and 14. Its value is below $2n'$.

- q' is the control variable for the loop between lines 4 and 13, iterated over the whole set of states. Hence, it can store one of n' possible different values.
- i is the control variable for the loop between lines 5 and 12, running from 1 to $m \leqslant n'$. Hence, its value does not exceed n'.
- q and q_{prev} contain some states reached by traversals. They clearly can store n' possible different values.
- $\tilde{t}$ and $\tilde{q}$ are local variables, the former used as a counter in the subroutine *simulation*, ranging from 0 to $t < 2n'$, the latter inside *reach*, used to simulate input traversals, containing some state, and hence storing one of n' possible values. Since $\tilde{t}$ and $\tilde{q}$ are never used simultaneously, they can be replaced by a single global variable $\tilde{x}$, capable of containing one of $2n'$ different values.

It is clear that all this information can be accommodated in $O((n')^8)$ states. Thus, a quasi-sweeping 2nfa A with $n' \leqslant 2n+2$ states can be replaced by an equivalent 2nfa that is quasi-sweeping, self-verifying, and halting, with $O(n^8)$ states.

Using Theorem 4.1, we can make any unary automaton quasi-sweeping, and this enables us to simulate an arbitrary n-state unary 2nfa. However, the new machine may disagree with the original one on "short" inputs, namely of length not exceeding $5n^2$.

The problem of a finite number of "short" inputs can be solved easily. In an initial phase, consisting of a single left-to-right traversal followed by a single right-to-left traversal, accept or reject all inputs of length not exceeding $5n^2$. This can be done deterministically, with $O(n^2)$ states. Then, if the input length exceeds $5n^2$, simulate the qsnfa A of Theorem 4.1 by inductive counting, with $O(n^8)$ states. This gives:

Theorem 4.2. *Each n-state unary 2nfa A can be replaced by an equivalent $O(n^8)$-state halting qssvfa A'.*

As pointed out in Section 2, by simply exchanging accepting with rejecting states in the self-verifying A' of Theorem 4.2, we get a machine for the complement of $L(A')$.

Corollary 4.3. *For each n-state unary 2nfa A, there exists an $O(n^8)$-state 2nfa A' accepting the complement of $L(A)$. Moreover, A' is quasi-sweeping, self-verifying, and halting.*

5. Simulation by unary probabilistic machines

Theorem 4.2 and Corollary 4.3 allow us to easily draw some further consequences. In the case of unary two-way automata, the reduction of nondeterminism to self-verifying nondeterminism can go down one more level, to Las Vegas automata (2lvfa, see Definition 2.1), still with only an $O(n^8)$-state penalty.

For the size (the number of states) of one-way finite automata, a polynomial relation between determinism and Las Vegas has been established [8], which implies an exponential gap between Las Vegas and nondeterminism. In the case of two-way automata, we do not know whether there is a polynomial relation between determinism and Las Vegas, or between Las Vegas and nondeterminism. However, the relation between self-verifying nondeterminism and Las Vegas was shown to be linear:

Theorem 5.1 [7, Thm. 1]**.** *Each n-state 2svfa A can be replaced by an equivalent $O(n)$-state 2lvfa A'.*

Combining Theorems 4.2 and 5.1, we get that, in the case of unary two-way automata, even unrestricted nondeterminism and Las Vegas are polynomially related:

Theorem 5.2. *Each n-state unary 2nfa A can be replaced by an equivalent $O(n^8)$-state 2lvfa A'.*

This might give additional evidence that nondeterministic two-way automata, when restricted to unary inputs, are not as powerful as they might look at first glance. Compare also with Theorem 4 in [4], where a subexponential (though not polynomial) simulation of unary 2nfa's by 2dfa's is presented, with a $2^{O(\log^2 n)}$-state penalty.

6. Concluding remarks

We have shown that, for deterministic two-way finite automata, the construction of an automaton for the complement of the language accepted by the original machine requires only a linear increase in the number of states. For nondeterministic two-way automata, when restricted to unary input alphabet, this relationship is polynomial.

We were not able to resolve the problem of complement for 2nfa's in the general (nonunary) case. We conjecture that there is no polynomial complementation of 2nfa's. Notice that a positive answer in this sense would immediately prove the most famous conjecture in this field, posed by Sakoda and Sipser in 1978 [19]. They ask whether the simulation of 2nfa's by 2dfa's is polynomial in the number of states. It is generally conjectured that the answer is negative, so we can formulate this open problem as follows:

(a) Prove that there exists an exponential gap (or at least superpolynomial, in the number of states) between nondeterministic and deterministic two-way finite automata.

As recalled in Section 1, proving (a) by the use of polynomially long strings would in turn prove L $\neq$ NL, by [1], a long-standing and fundamental open question. Let us now present two related open problems:

(b) Prove that there exists an exponential, or at least superpolynomial, gap between self-verifying and deterministic two-way finite automata.
(c) Prove that there exists an exponential, or at least superpolynomial, gap between nondeterministic and self-verifying two-way finite automata.

Clearly, an argument for (b) would immediately prove (a), since a self-verifying machine is also a nondeterministic machine. The same holds for (c), since, by Corollary 3.4, we can make each 2dfa halting, with $O(n)$ states. Then, by stopping in an extra rejecting state q_f^r (if the original machine does not accept), we make the given deterministic machine self-verifying. Yet, the problem (c) is equivalent to the following:

(d) Prove that there exists an exponential, or at least superpolynomial, gap between 2nfa's and 2nfa's accepting the complement.

It is not too hard to see that (d) is equivalent to (c). Suppose that some $p(n)$ states are sufficient to make an arbitrary 2nfa self-verifying. Then $p(n)$ states are sufficient to build a 2nfa for the complement; such a 2nfa is the 2svfa where the original rejecting states are accepting and, at the same time, the accepting states are made rejecting. (Both machines have the same set of don't-know states.) Conversely, if there exists a polynomial complementation of 2nfa's, using $p(n)$ states, then $n+p(n)$ states are sufficient to make an arbitrary 2nfa self-verifying. We simply combine A_1 and A_2, the respective machines for the language and its complement, into a single machine. The set of accepting states of the new machine is exactly the set of accepting states of A_1, the set of its rejecting states exactly the set of accepting states of A_2. The new machine starts in $q_{0,1}$, the initial state of A_1, however, it can switch to the initial state of A_2 at the left endmarker, by $(q_{0,2}, 0) \in \delta(q_{0,1}, \vdash)$.

Thus, the problem (d) is equivalent to (c).

Finally, taking into account the upper bounds presented in Corollary 3.2 and Theorem 3.3, the following open problem arises:

(e) Prove (or disprove) the existence of *any* gap between 2dfa's and 2dfa's accepting the complement.

By Corollary 3.2, it follows that a minimized 2dfa for the witness language (if it exists) must reject some inputs by getting into an infinite loop.

Acknowledgment

The authors thank an anonymous referee for her/his helpful comments and remarks.

References

[1] P. Berman, A. Lingas, On the complexity of regular languages in terms of finite automata. Tech. Report 304, Polish Academy of Sciences, 1977.
[2] M. Dietzelfelbinger, M. Kutylowski, R. Reischuk, Exact lower bounds for computing Boolean functions on CREW PRAMs, J. Comput. Syst. Sci. 48 (1994) 231–254.
[3] V. Geffert, Tally versions of the Savitch and Immerman–Szelepcsényi theorems for sublogarithmic space, SIAM J. Comput. 22 (1993) 102–113.
[4] V. Geffert, C. Mereghetti, G. Pighizzini, Converting two-way nondeterministic automata into simpler automata, Theoret. Comput. Sci. 295 (2003) 189–203.
[5] J. Goldstine, M. Kappes, C. Kintala, H. Leung, A. Malcher, D. Wotschke, Descriptional complexity of machines with limited resources, J. Universal Comput. Sci. 8 (2002) 193–234.
[6] J. Hopcroft, J. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, 1979.
[7] J. Hromkovič, G. Schnitger, On the power of Las Vegas II: two-way finite automata, Theoret. Comput. Sci. 262 (2001) 1–24.
[8] J. Hromkovič, G. Schnitger, On the power of Las Vegas for one-way communication complexity, OBDDs, and finite automata, Inform. Comput. 169 (2001) 284–296.
[9] N. Immerman, Nondeterministic space is closed under complementation, SIAM J. Comput. 17 (1988) 935–938.
[10] J. Jirásek, G. Jirásková, A. Szabari, State complexity of concatenation and complementation, Int. J. Found. Comput. Sci. 16 (2005) 511–529.
[11] C. Kapoutsis, Deterministic moles cannot solve liveness, in: Proc 7th Int. Workshop on Descriptional Complexity of Formal Systems (DCFS 2005), University of St. Milano, Como, Italy, 2005, pp. 194–205.
[12] A. Kondacs, J. Watrous, On the power of quantum finite state automata, in: Proc. 38th Symp. Found. Comp. Sci. (FOCS 1997), IEEE Computer Society Press, 1997, pp. 66–75.
[13] F. Mera, G. Pighizzini, Complementing unary nondeterministic automata, Theoret. Comput. Sci. 330 (2005) 349–360.
[14] C. Mereghetti, G. Pighizzini, Two-way automata simulations and unary languages, J. Aut. Lang. Combin. 5 (2000) 287–300.
[15] A. Meyer, M. Fischer, Economy of description by automata, grammars, and formal systems, in: Proc. 12th Ann. IEEE Symp. on Switching and Automata Theory, 1971, pp. 188–191.
[16] C. Moore, J. Crutchfield, Quantum automata and quantum grammars, Theoret. Comput. Sci. 237 (2000) 275–306.
[17] F. Moore, On the bounds for state-set size in the proofs of equivalence between deterministic, nondeterministic and two-way finite automata by deterministic automata, IEEE Trans. Comput. C-20 (1971) 1211–1219.
[18] M.O. Rabin, Probabilistic automata, Inform. Control 6 (1963) 230–245.
[19] W. Sakoda, M. Sipser, Nondeterminism and the size of two-way finite automata, in: Proc. 10th ACM Symp. Theory of Computing, 1978, pp. 275–286.
[20] M. Sipser, Lower bounds on the size of sweeping automata, J. Comput. Syst. Sci. 21 (1980) 195–202.
[21] M. Sipser, Halting space bounded computations, Theoret. Comput. Sci. 10 (1980) 335–338.
[22] R. Szelepcsényi, The method of forced enumeration for nondeterministic automata, Acta Inform. 26 (1988) 279–284.
[23] M.Y. Vardi, A note on the reduction of two-way automata to one-way automata, Inform. Process. Lett. 30 (1989) 261–264.