

# Linear-Time Limited Automata<sup>☆</sup>

Bruno Guillon<sup>a,\*</sup>, Luca Prigioniero<sup>b,\*\*</sup>

<sup>a</sup>*INRIA Lille, France*

<sup>b</sup>*Dipartimento di Informatica, Università degli Studi di Milano, Italy*

---

## Abstract

The time complexity of 1-limited automata is investigated from a descriptive complexity view point. Though the model recognizes regular languages only, it may use quadratic time in the input length. We show that, with a polynomial increase in size and preserving determinism, each 1-limited automaton can be transformed into a linear-time equivalent one. We also obtain polynomial transformations into related models, including weight-reducing Hennie machines (*i. e.*, one-tape Turing machines syntactically forced to operate in linear-time), and we show exponential gaps for the converse transformations in the deterministic case.

*Keywords:* regular languages, limited automata, descriptive complexity

---

## 1. Introduction

One classical topic of computer science is the investigation of computational models operating under restrictions. Finite automata or pushdown automata, for instance, can be considered as particular Turing machines in which the access to memory storage is limited. Other kinds of restrictions follow from finer analyses of the computational resources an abstract device requires to recognize certain languages. For example, in the case of Turing machines, classical

---

<sup>☆</sup>This is an extended version of results presented in [1]

\*Principal corresponding author

\*\*Corresponding author

*Email addresses:* `bruno.guillon@inria.fr` (Bruno Guillon), `prigioniero@di.unimi.it` (Luca Prigioniero)

complexity classes such as P, NP, LOGSPACE, *etc.* are defined by introducing a limit on the amount of resources, namely time or space, at disposal of the model.

Usually, such limitations reduce the expressive power. For instance, it is well-known that one-tape nondeterministic Turing machines operating within a space bounded by the length of the input, namely *linear bounded automata*, capture exactly the class of *context-sensitive languages*, see *e.g.* [2]. Phenomena like this, where limiting an abstract model reduces its expressiveness to the level of some standard class, are of great interest, as they provide alternative characterizations of language classes.

Another example of this kind has been observed by Hennie in 1965. He indeed proved that deterministic one-tape Turing machines operating in *linear time* recognize exactly the class of *regular languages* [3]. The result has later been extended to the nondeterministic case [4, 5]. Here, operating in linear time means that every computation has length linearly bounded in the input length. In particular, linear-time machines are necessarily halting – see [5] for investigations of alternative linear time restrictions. By *Hennie machine* we refer to nondeterministic linear bounded automaton operating in linear time. The above-mentioned result implies that every Hennie machine is equivalent to some finite automaton. From the opposite point of view, this means that providing *two-way finite automata* with the ability to overwrite the tape cells does not extend the expressiveness of the model, as long as the time is linearly bounded in the length of the input.

However, Průša showed that it is undecidable given a deterministic linear bounded automaton to check whether it works in linear time over all input strings, namely, whether it is actually a Hennie machine [6]. To avoid this drawback, he proposed a variant of Hennie machine, called *weight-reducing Hennie machine*, in which the time limitation is syntactic. In this model, each visit to a cell should overwrite its content with a symbol in a decreasing way, with respect to some fixed order on the working alphabet. As a consequence, the number of

visits of a cell by the head is bounded by some constant (*i.e.*, not depending on the input length) hence the device works in linear time over every input string.

By contrast to Hennie machines, the *d-scan limited automata* (or simply *d-limited automata*) introduced by Hibbard, restrict nondeterministic linear bounded automata by allowing overwriting of each tape cell during its first  $d$  visits only, for some fixed  $d \geq 0$  [7]. Contrary to weight-reducing Hennie machines, the head is still allowed to visit a cell after the  $d$ -th visit, but it cannot rewrite the content anymore. This allows to use super-linear time. Hence, *limited automata* (namely,  $d$ -limited automata for some  $d$ ) live midway between linear bounded automata and weight-reducing Hennie machines. Hibbard proved that, for each  $d \geq 2$ ,  $d$ -limited automata recognize exactly the class of *context-free languages*. He furthermore showed the existence of an infinite hierarchy of deterministic  $d$ -limited automata, whose first level (*i.e.*, corresponding to deterministic 2-limited automata) has been later proved to coincide with the class of *deterministic context-free languages* [8]. (See [9] and references therein for further connections between limited automata and context-free languages.)

Clearly, 0-limited automata are no more than two-way finite automata hence characterize the class of regular languages. Wagner and Wechsung extended this result to the case  $d = 1$ : 1-limited automata recognize exactly the class of regular languages [10]. From that point, the question of the cost of their simulation by classical finite automata has been studied by Pighizzini and Pisoni in [11], where a tight doubly-exponential simulation by deterministic one-way finite automata is proved. This cost reduces to a single exponential when starting from a deterministic 1-limited automaton. Moreover, an exponential lower bound has been obtained in [12], for the simulation of deterministic 1-limited automata by nondeterministic two-way finite automata.

Like  $d$ -limited automata, 1-limited automata can operate in super-linear time (*cf.* Example 1). This contrasts with Hennie machines which operate in linear time by definition. The question we address in this paper is whether this ability of 1-limited automata with respect to Hennie machines yields a gap between

the two models in terms of the size of their representations.

We show that, with a polynomial increase in size, each 1-limited automaton can be transformed into an equivalent linear-time 1-limited automaton, or, alternatively, into a weight-reducing Hennie machine. Furthermore, we are able to obtain a deterministic device when the original machine is deterministic. We also show that the 1-limited automata resulting from our constructions have a special structure that can be exploited in order to obtain equivalent 1-limited automata in which an initial memoryless phase overwrites each tape cell, *i.e.*, the device initially performs a nondeterministic left-to-right pass over the tape during which all the cells are independently overwritten. Similar behaviors have been considered in the context of *regular transduction*, because of their correspondence with global existential quantification in *monadic second order logic*, see [13] in which the authors define an operation called *common guess* corresponding to a nondeterministic memoryless overwriting of the tape. Hence, as a consequence of our main result, each 1-limited automaton can be simulated by a *two-way automaton with common guess* of polynomial size. It follows that *reversing* a 1-limited automaton, *i.e.*, transforming it into another one recognizing the reverse of its accepted language, has polynomial cost only. This fails in the deterministic case, for which we exhibit an exponential lower bound. As a consequence, we obtain exponential lower bounds for the simulation of deterministic weight-reducing Hennie machines or deterministic two-way automata with common guess by deterministic 1-limited automata. The results are summarized in Figure 1.

The paper is organized as follows. In Section 2 are gathered the main definitions and notations needed in the subsequent sections. The main constructions used for proving our results are detailed in Section 3, while the results are finally presented in Section 4.

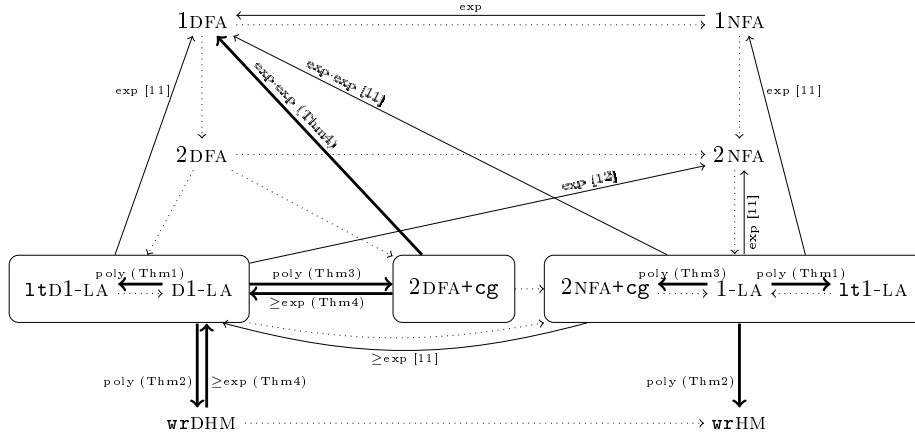


Figure 1: Relationships between the main models studied in the paper. Here, **1t** and **wr** mean linear-time and weight-reducing, while **D1-LA** and **(D)HM** stand for deterministic 1-LA and (deterministic) Hennie machine, respectively. Deterministic and nondeterministic two-way automata with common guess are denoted by **2DFA+cg** and **2NFA+cg**. Dotted arrows indicate trivial connections while thick arrows indicate our results.

## 2. Preliminaries

In this section we recall some basic definitions and notations useful in the paper. In particular, we assume the reader familiar with notions from formal languages and automata theory (see, *e.g.*, [2]). Given a set  $S$ ,  $\#S$  denotes its cardinality and  $2^S$  the family of all its subsets. Given an alphabet  $\Sigma$ , we denote by  $|w|$  the length of a string  $w \in \Sigma^*$ , by  $w^R$  the reversal of  $w$  and by  $\varepsilon$  the empty string. For a language  $L \subseteq \Sigma^*$ ,  $L^R$  denotes the *reversal* of  $L$ , namely  $L^R = \{w^R \mid w \in L\}$ .

**Definition 1.** A **two-way nondeterministic finite automaton (2NFA)** is a quintuple  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ , where  $Q$  is a finite set of states,  $\Sigma$  is a finite input alphabet,  $q_0 \in Q$  is the initial state,  $F \subseteq Q$  is a set of final states, and  $\delta : Q \times \Sigma_{\triangleright\triangleleft} \rightarrow 2^{Q \times \{-1, 0, +1\}}$  is a nondeterministic transition function where  $\Sigma_{\triangleright\triangleleft}$  denotes the set  $\Sigma \cup \{\triangleright, \triangleleft\}$  with the two special symbols  $\triangleright, \triangleleft \notin \Sigma$  respectively called the **left** and the **right** endmarkers.

The input is written on the tape surrounded by the two endmarkers, the left endmarker being at the position zero. Hence, on input  $w$ , the right endmarker

is at position  $|w| + 1$ . In one move,  $\mathcal{A}$  reads an input symbol, changes its state, and moves the input head one position backward, forward or keeps it in position depending on whether  $\delta$  returns  $-1$ ,  $+1$  or  $0$ , respectively. Furthermore, the head cannot violate the endmarkers, except at the end of computation, to accept the input, as now explained. The machine *accepts* the input, if there exists a computation path starting from the initial state  $q_0$  with the head on the first tape cell (*i.e.*, scanning the left endmarker) and ending in a final state  $q \in F$  after violating the right endmarker. The language accepted by  $\mathcal{A}$  is denoted by  $L(\mathcal{A})$ . A 2NFA  $\mathcal{A}$  is said to be *deterministic* (2DFA), whenever  $\#\delta(q, \sigma) \leq 1$ , for any  $q \in Q$  and  $\sigma \in \Sigma_{\triangleright\triangleleft}$ . It is called *one-way* if its head can never move backward, *i.e.*, if no transition returns  $-1$ . By 1NFAs and 1DFAs we denote one-way nondeterministic and deterministic finite automata, respectively.

We now introduce the main model we are interested in. A 1-limited automaton is a linear bounded automaton which can rewrite the content of each tape cell in the first visit only. Formally:

**Definition 2.** A *1-limited automaton* (1-LA) is a tuple  $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, F)$ , where  $Q$ ,  $\Sigma$ ,  $q_0$  and  $F$  are defined as for 2NFAs,  $\Gamma$  is a finite *working alphabet* such that  $\Sigma \subset \Gamma$ ,  $\delta : Q \times \Gamma_{\triangleright\triangleleft} \rightarrow 2^{Q \times \Gamma_{\triangleright\triangleleft} \times \{-1, 0, +1\}}$  is the *nondeterministic transition function* where  $\Gamma_{\triangleright\triangleleft}$  denotes the set  $\Gamma \cup \{\triangleright, \triangleleft\}$  with  $\triangleright, \triangleleft \notin \Gamma$  the left and the right endmarkers as for 2NFAs. Moreover, for each transition  $(q, \gamma, d) \in \delta(p, \sigma)$ , we have  $\gamma \in \Gamma_{\triangleright\triangleleft} \setminus \Sigma$  and if  $\sigma \in \Gamma_{\triangleright\triangleleft} \setminus \Sigma$  then  $\gamma = \sigma$ .

In one move, according to  $\delta$ ,  $\mathcal{A}$  reads a symbol from the tape, changes its state, replaces the symbol just read by a new symbol, and moves its head one position backward or forward or keeps it in place. However, replacing symbols is subject to some restrictions, which, essentially, allow to modify the content of a cell during the first visit only. Technically, symbols from  $\Sigma$  shall be replaced by symbols from  $\Gamma \setminus \Sigma$ , while symbols from  $\Gamma_{\triangleright\triangleleft} \setminus \Sigma$  are never overwritten. In particular, at any time, both special symbols  $\triangleright$  and  $\triangleleft$  occur exactly once on the tape and exactly at the respective left and right boundaries. *Acceptance* for 1-LAs as well as *deterministic 1-LAs* are defined exactly as for 2NFAs, and

the language accepted by a given 1-LA  $\mathcal{A}$  is denoted by  $L(\mathcal{A})$ .

The following result, which is instrumental for our later proofs, gives a simpler form of 1-LAs.

**Lemma 1.** *For each  $n$ -state 1-LA, there exists an equivalent  $3n$ -state 1-LA using the same working alphabet, which performs stationary moves exactly when rewriting a cell content. Furthermore, the conversion preserves determinism.*

*Proof.* First, by using standard techniques, each sequence of stationary moves followed by a nonstationary move can be replaced by a nonstationary transition. This operation does not increase the size of the 1-LA. Since in every accepting computation the last transition performed by any 1-LA is a right move from the rightmost cell of the tape, this modification eliminates all the stationary moves.

Second, we can split every rewriting step into two steps: a first stationary step during which the input cell is rewritten, followed by a second read-only nonstationary step. This yields a linear increase of the size of the device only. More precisely, for each state  $q \in Q$  and each direction  $d \in \{-1, +1\}$ , we create a new copy  $(q, d) \notin Q$  of  $q$ , whose interpretation is to delay the head move  $d$ . Then, each rewriting transition  $(q, \gamma, d) \in \delta(p, \sigma)$  is replaced by two transitions  $((q, d), \gamma, 0)$  from  $p$  on  $\sigma$  and  $(q, \gamma, d)$  from  $(q, d)$  on  $\gamma$ .  $\square$

### Configurations and computations

For each of the above-defined models, a *configuration* is represented as a string  $z \bullet p \bullet z'$ , meaning that  $p$  is the current state,  $zz' \in \triangleright \Pi^* \triangleleft$  is the content of the tape (here  $\Pi$  denotes the alphabet  $\Sigma$  or  $\Gamma$  depending on the model under consideration) and the head is scanning the first symbol of  $z'$ . The transition relation between configurations is denoted by  $\vdash$ , and its reflexive-transitive closure by  $\vdash^*$ . Notice that, in case  $|z'| = 0$ , the machine has reached the end of the computation. We also represent *partial configurations* as  $u \bullet p \bullet v$ , where  $p$  is the current state and  $uv \in \{\varepsilon, \triangleright\} \Pi^* \{\varepsilon, \triangleleft\}$  is a factor of the tape content. The relations  $\vdash$  and  $\vdash^*$  naturally extend onto partial configurations.

### *Size of models*

For each model under consideration, we evaluate its size as the total number of symbols used to describe it. Hence, under standard representation, the *sizes* of an  $n$ -state 2NFAs and of  $n$ -state 1-LAs are respectively  $O(n^2\#\Sigma)$  and  $O(n^2\#\Gamma)$ .

**Example 1.** We consider the language

$$L_n = \{x_0x_1 \cdots x_k \mid k \in \mathbb{N}, \text{ for each } i: x_i \in \{a, b\}^n, \text{ for some } j \neq 0: x_j = x_0\}.$$

A deterministic 1-LA  $\mathcal{A}_n$  may recognize  $L_n$  as follows. It first scans the factor  $x_0$ , overwriting each input symbol with a marked copy. Then,  $\mathcal{A}_n$  repeats a subroutine which overwrites a factor  $x_i$  with some fixed symbol  $\sharp$ , while checking in the meantime whether  $x_i$  equals  $x_0$  or not. This can be achieved as follows. Before overwriting the  $j$ -th symbol of  $x_i$ , first,  $\mathcal{A}_n$ , with the help of a counter modulo  $n$ , moves the head leftward to the position  $j$  of  $x_0$  and stores the unmarked scanned symbol  $\sigma$  in its finite control; second, it moves the head rightward until reaching the position  $j$  of  $x_i$ , namely, the leftmost position that has not been overwritten so far. At this point,  $\mathcal{A}_n$  compares the scanned symbol (*i.e.*, the  $j$ -th symbol of  $x_i$ ) with  $\sigma$  (*i.e.*, the  $j$ -th symbol of  $x_0$ ). By setting a Boolean flag to true when complete factor  $x_i$  has matched  $x_0$  and finally checking that the input string has length multiple of  $n$ ,  $\mathcal{A}_n$  can decide the membership of the input to  $L_n$ .

It is possible to implement  $\mathcal{A}_n$  with a number of states linear in  $n$  and  $\#\Sigma + 1$  working symbols. Since for each position of a factor  $x_i$ ,  $i > 0$ , the head has to move back to the factor  $x_0$ , we observe that  $\mathcal{A}_n$  works in quadratic time in the length of the input string.

### **3. Linear-Time Simulations of 1-LAs**

If a linear-space Turing machine can visit a tape cell only a constant number of times, it necessarily works in linear time. Conversely, Turing machines working in linear time (*i.e.*, Hennie machines), have been shown to visit each tape



cell only a constant number of times during a computation [3]. This contrasts with the case of 1-LAs, which can use quadratic time, as shown in Example 1. However, our main contribution states that, with a polynomial increase in size of the model, we can recover the above property, and therefore obtain equivalent 1-LAs working in linear time.

### 3.1. Main ingredients

#### 3.1.1. Local window space bound

The key idea to obtain a linear time bound, is to ensure that, in any computation, the simulating device works on a virtual window of fixed size that is shifted along the tape in a one-way manner. More precisely, in the computations of our simulating 1-LAs, there exists a constant  $K$  not depending on the input length, such that, for any two tape cells at distance  $K$ , the leftmost one cannot be visited after having visited the rightmost one. In this way, it is possible to bound the number of visits of each cell.

In our simulation we divide the input word into blocks of some fixed length  $\ell$ , given by some polynomial in the number  $n$  of states of the simulated 1-LA. Then, our virtual window covers two successive blocks, *i.e.*,  $K = 2\ell$ . The length  $\ell$  is chosen in such a way that, once overwritten, a block on the tape may contain the sufficient information for recovering the behaviors of the simulated machine that may occur on the left of the window. Describing and storing this information is the purpose of the following subsection.

#### 3.1.2. Shepherdson tables

In [11], the authors presented a construction to simulate any 1-LA  $\mathcal{A}$  by a finite automaton  $\mathcal{B}$ , using classic ideas from the simulation of two-way automata by one-way automata [14]. The main ingredient was to store in the finite control of  $\mathcal{B}$ , a “transition table” describing the possible behaviors of  $\mathcal{A}$  that may occur to the left of the current head position. Since the part of the tape to the left of the current head position has necessarily already been visited, its “frozen” content belongs to  $\triangleright(\Gamma \setminus \Sigma)^* \cup \{\varepsilon\}$ . Hence, the above-mentioned behaviors to the left of the current head position are read-only computations. To represent them, for

each word  $z'X \in \triangleright(\Gamma \setminus \Sigma)^*$  with  $|X| = 1$ , we consider a relation  $\tau_{z'X} \subseteq Q \times Q$ , where  $Q$  denotes the set of states of  $\mathcal{A}$ . A pair  $(p, q)$  belongs to  $\tau_{z'X}$  if and only if, starting from state  $p$  with the head scanning the last symbol of  $z'X$ ,  $\mathcal{A}$  may reach state  $q$  one cell to the right of  $z'X$ . Formally,

$$\tau_{z'X} = \{(p, q) \mid z' \bullet p \bullet X \vdash^* z'X \bullet q\}.$$

With the information of  $\tau_{z'X}$ ,  $\mathcal{B}$  has no need to read the part of the tape containing  $z'X$ , that is, to move its head leftward. Furthermore, given a string  $x \in (\Gamma \setminus \Sigma)^*$ , we can construct  $\tau_{z'Xx}$  from  $\tau_{z'X}$  by scanning  $x$ . For convenience, we set  $\tau_\varepsilon = \emptyset$ .

We denote by  $n$  the cardinality of  $Q$ . In [11], the table of size  $n^2$  corresponding to the relation  $\tau_{z'X}$  was stored in the finite control of the simulating 1NFA  $\mathcal{B}$  and it was updated at each step. This yielded an exponential number of states for storing the  $2^{n^2}$  possible tables thus implying an exponential size of  $\mathcal{B}$  with respect to  $\mathcal{A}$ . This blowup was shown to be necessary in the worst case for the considered simulation.

Here, as our simulating device is a 1-LA, we take advantage of its ability to write on the tape. We indeed store the table  $\tau_{z'X}$  onto the  $n^2$  cells following the last position of  $z'X$ . Formally, fixing a bijection  $\mu$  from  $\{0, \dots, n^2 - 1\}$  to  $Q \times Q$ , the  $i$ -th cell of the portion storing the table  $\tau_{z'X}$  will contain 1 if  $\mu(i)$  belongs to  $\tau_{z'X}$ , and 0 otherwise. As a consequence, we do not store all the tables corresponding to each tape position but a subcollection of them. More precisely, we only store tables  $\tau_{z'X}$  for tape content prefixes  $z'X$  of length multiple of  $n^2$ . Thus, updating the tables should be done block by block rather than cell by cell, for a decomposition of the input into blocks of length  $n^2$ . (We consider the cell containing the left endmarker as a complete block, while the last block containing the right endmarker may be shorter than  $n^2$ .)

When  $\mathcal{A}$  is deterministic,  $\tau_{z'X}$  is a partial<sup>1</sup> function from  $Q$  to  $Q$ . In this

---

<sup>1</sup>In the deterministic case, the image associated with  $p$  by  $\tau_{z'X}$  is undefined if one of the two following cases of the computation starting in  $z' \bullet p \bullet X$  occurs: either, after a finite

case it is possible to improve the above-described construction by storing the tables  $\tau_{z'X}$  on the  $n$  cells following the last position of  $z'X$ . The input is therefore decomposed into blocks of length  $n$  rather than  $n^2$ . However, the alphabet used to store the table has size  $n + 1$  rather than 2. Indeed, the  $i$ -th cell of the portion storing the table will contain the image of the  $i$ -th state of  $\mathcal{A}$  by  $\tau_{z'X}$  if defined, or a special symbol  $\perp \notin Q$  otherwise.

### 3.2. Construction of the simulating 1-LA $\mathcal{A}'$

Our simulation combines the two ideas discussed previously, by storing a subcollection of the Shepherdson tables on the tape. We actually present several simulations transforming 1-LAs into equivalent linear-time 1-LAs. The most general one produces a nondeterministic 1-LA from a nondeterministic 1-LA. The other ones produce a deterministic 1-LA from a deterministic 1-LA. The various constructions are very similar and differ only in some basic routines and in the encoding of the Shepherdson tables. We first present their common global structure and then we specify their differences, when detailing the low-level implementation of the basic operations and subroutines used for the simulation in Section 3.2.3. To this end, we now fix some convenient notations.

Let  $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, F)$  be the source 1-LA. By Lemma 1, modulo a linear size increase, we suppose that  $\mathcal{A}$  performs stationary moves exactly when overwriting a cell content. Our goal is to build a linear-time 1-LA  $\mathcal{A}' = (Q', \Sigma, \Gamma', \delta', q'_0, F')$  equivalent to  $\mathcal{A}$ , which has polynomial size with respect to the size of  $\mathcal{A}$ .

Let  $\ell$  denote the size of the blocks in the tape decomposition discussed above, and let  $T$  denote the set of symbols used to encode the Shepherdson tables on tape. Formally, either  $\ell = n^2$  and  $T = \{0, 1\}$ , or, possibly if  $\mathcal{A}$  is deterministic,  $\ell = n$  and  $T = Q_\perp$  where  $Q_\perp = Q \cup \{\perp\}$  for  $\perp$  a symbol not belonging to  $Q$ . Moreover, we fix a mapping  $\nu$  from  $\{0, \dots, \ell - 1\}$  to  $Q$  defined for each index  $i \in \{0, \dots, \ell - 1\}$  as follows:

---

number of steps, no successive transition is defined (incompleteness of  $\mathcal{A}$ ), or the computation eventually enters a deterministic loop (non-haltingness of  $\mathcal{A}$ ).

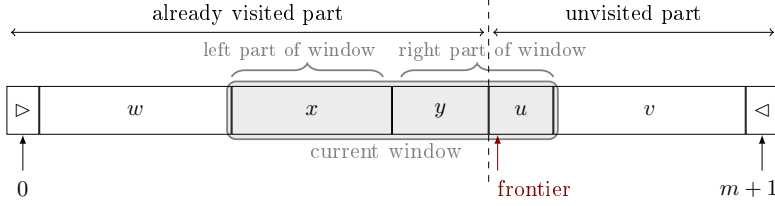


Figure 2: Typical description of the window during a computation of  $\mathcal{A}$ :  $m$  denotes the length of the input word, the current frontier occurs in the right block as first position of  $u$ ,  $w \in (\Gamma \setminus \Sigma)^\ell$ ,  $x \in (\Gamma \setminus \Sigma)^\ell$ ,  $y \in (\Gamma \setminus \Sigma)^*$ ,  $u \in \Sigma^+$  with  $|yu| = \ell$ , and  $v \in \Sigma^*$ .

- if  $\ell = n^2$  then  $\nu(i)$  is the state  $p$  such that  $\mu(i) = (p, q)$  for some state  $q$  (recall that  $\mu$  is a fixed bijection from  $\{0, \dots, n^2 - 1\}$  to  $Q^2$ );
- if  $\ell = n$  then  $\nu(i) = q_i$ , assuming  $Q = \{q_0, \dots, q_{n-1}\}$ .

During a computation of our simulating 1-LA  $\mathcal{A}'$ , the frozen content of the tape can be viewed as divided into two tracks: the first track contains the symbols overwritten by  $\mathcal{A}$  in the simulated computation; the second track contains the encoding of the Shepherdson tables  $\tau_{z'X}$ . We thus fix the set of working symbols to be the product of  $\Gamma \setminus \Sigma$  and  $T$ , *i.e.*,

$$\Gamma' = ((\Gamma \setminus \Sigma) \times T) \cup \Sigma.$$

As previously explained, the behavior of  $\mathcal{A}'$  will be locally restricted to a window of bounded width. At any time in a computation of  $\mathcal{A}$  we consider a virtual window which covers two successive blocks in the tape decomposition described above. The right block covered by the window contains the leftmost cell that has not been visited so far, to which we refer as current *frontier*. The content  $x$  of the left block covered by the window belongs to  $(\Gamma \setminus \Sigma)^\ell \cup \{\triangleright\}$ . The content of the right block covered by the window can be decomposed into  $yu$  with  $y \in (\Gamma \setminus \Sigma)^*$  and  $u \in \Sigma^+ \cup \Sigma^* \triangleleft$  such that  $|yu| = \ell$  unless, possibly,  $\triangleleft$  occurs in  $u$ , in which case  $|yu| \leq \ell$ . The frontier is on the first position of  $u$ . A typical situation is depicted in Figure 2.

In order to simulate  $\mathcal{A}$ , the linear-time 1-LA  $\mathcal{A}'$  overwrites each block with a word  $\tilde{x} \in (\Gamma' \setminus \Sigma)^\ell$  whose projection on  $\Gamma \setminus \Sigma$  is the word  $x$  written by  $\mathcal{A}$  on

the corresponding block in the simulated computation, and the projection on  $T$  is exactly the encoding of the table  $\tau_z$ , where  $z$  is the content of the tape to the left of the corresponding block in the simulated computation. (In Figure 2,  $z = \triangleright w$  when considering the left block covered by the window, whose frozen content is  $x$ .) Roughly, in the simulating computation, when the frontier occurs at the first position of a block,  $\mathcal{A}'$  has to fill this block, cell by cell, with the encoding of  $\tau_{zx}$ , where  $x$  (resp.  $z$ ) is the projection on  $\Gamma \setminus \Sigma$  of the content  $\tilde{x}$  of the preceding block which is covered by the window (resp. of the content  $\tilde{z}$  of the tape to the left of the current window). To this end, it has read-only access to the left block, whose content  $\tilde{x}$  gathers all the required information, namely  $\tau_z$  and  $x$ . In parallel,  $\mathcal{A}'$  should also recover the simulated computation of  $\mathcal{A}$ . As soon as the right block is completely filled, the window is shifted to the right, in such a way that it covers the block just treated (as left part) and its successor (as right part).

### 3.2.1. Auxiliary procedures `readFromTable` and `simulateLeft`

Our simulation uses two subroutines, `readFromTable` and `simulateLeft`, which are called in order to recover some value from  $\tau_z$ , where  $z$  is a tape content prefix in the simulated computation. Using the above-given notations (see, e.g., Figure 2), we suppose that the virtual window covers two successive blocks, the left one containing  $\tilde{x} \in (\Gamma' \setminus \Sigma)^\ell \cup \{\triangleright\}$ , and the right one being partially filled with a prefix  $\tilde{y} \in (\Gamma' \setminus \Sigma)^*$  of length less than  $\ell$ . We denote by  $z \in \triangleright \left( (\Gamma \setminus \Sigma)^\ell \right)^* \cup \{\varepsilon\}$  the tape content to the left of the window in the simulated computation.

`readFromTable` starts from and ends in the first position of the block containing  $\tilde{x}$  with a state  $p$  given as argument and returns a state  $q$  such that  $(p, q) \in \tau_z$ . Alternatively, it may return the error symbol  $\perp$  if no such  $q$  exists or if its internal computation fails. Nevertheless, for each state  $q$  such that  $(p, q) \in \tau_z$ , the procedure may return  $q$ . During its computation, the subroutine visits cells of the block containing  $\tilde{x}$  only.

**simulateLeft** starts from and ends in the last position of  $\tilde{x}$  (resp. of  $\tilde{y}$  if  $y \neq \varepsilon$ ) with a state  $p$  given as argument and returns a state  $q$  such that  $(p, q) \in \tau_{zx}$  (resp.  $(p, q) \in \tau_{zy}$ ). Alternatively, it may return the error symbol  $\perp$  if no such  $q$  exists or if its internal computation fails. Nevertheless, for each state  $q$  such that  $(p, q) \in \tau_{zx}$  (resp.  $(p, q) \in \tau_{zy}$ ), the procedure may return  $q$ . During its computation, the subroutine visits cells of the block containing  $\tilde{x}$  (resp. of the portion of the tape containing  $\tilde{x}\tilde{y}$ ) only.

When the simulated 1-LA is deterministic, it is possible to implement both subroutines in a deterministic way. The implementations of these subroutines are described in Section 3.2.3.

### 3.2.2. Main procedure

We now focus on the high-level description of the simulation, which is given in Procedure 1. By using a state component of size  $2\ell$ , named *relative position* and stored in a global variable **relativePosition**,  $\mathcal{A}'$  can store the exact position of its head relative to the current window. We represent it as a pair  $(i, s)$ , where  $i \in \{0, \dots, \ell - 1\}$  is the position in the scanned block of length  $\ell$  and  $s \in \{L, R\}$  is equal to L (resp. R) if the head is scanning a position in the left (resp. right) block of the window. We suppose that the component is updated at each head move. Using this component,  $\mathcal{A}'$  can avoid moving to the left of the current window. More precisely, from a relative position  $(0, L)$  (*i.e.*, the leftmost position covered by the window), when a backward move of  $\mathcal{A}$  from  $p$  to  $q$  has to be simulated,  $\mathcal{A}'$  calls the procedure **readFromTable** with argument  $q$ , in order to find a state  $r$  such that  $(q, r) \in \tau_z$ , where  $z$  is the content of the tape to the left of the window. Hence, it simulates not only the backward step from  $p$  to  $q$ , but also a complete computation segment to the left of the window, namely, it simulates  $z'X \bullet p \vdash z' \bullet q \bullet X \vdash^* z'X \bullet r$ , where  $z'X = z$ .

In addition to the relative position,  $\mathcal{A}'$  stores in a global variable, named **relativeFrontier**, the relative position of the current frontier, to which we refer as *relative frontier*. Since this position always occurs in the right block of the window, it is enough to represent it as an index  $\rho \in \{0, \dots, \ell - 1\}$ . Much

---

**Procedure 1: main**

---

```
/* the variables relativePosition and relativeFrontier are not indicated
   and are supposed to be automatically updated;
   in the following, “current symbol” designates the symbol
   currently read by the head */
1 frontierState  $\leftarrow q_0$ 
2 tableState  $\leftarrow \nu(\text{relativeFrontier})$ 
3 while “current symbol”  $\neq \triangleleft$  do
4   simulateLeft(frontierState)
5   if frontierState =  $\perp$  then REJECT
6   move the input head leftward until reaching position  $(\ell - 1, L)$ 
7   simulateLeft(tableState)
8   move the input head rightward until reaching position relativeFrontier - 1
9   move the input head one cell to the right
10  if “current symbol”  $\neq \triangleleft$  then
11    let  $(q, \gamma, 0) = \text{selectTransition}(\text{frontierState}, \text{“current symb”})$ 
12    frontierState  $\leftarrow q$ 
13    if frontierState =  $\perp$  then REJECT
14    write( $\gamma$ , tableState)
15    tableState  $\leftarrow \nu(\text{relativeFrontier})$ 
16  simulateLeft(frontierState)
17 if frontierState  $\in F$  then ACCEPT else REJECT
```

---

like the relative position component, we suppose that it is updated each time a cell is visited for the first time. Observe that such updates are increments modulo  $\ell$ . Incrementing  $\rho = \ell - 1$  means shifting the window by  $\ell$  cells to the right. In particular, this implies to updating the relative position by switching it from  $(\ell - 1, R)$  to  $(\ell - 1, L)$ .

Initially, the head is scanning the left endmarker, which is considered as the left block of the current window. Hence, the initial relative position and relative frontier are  $(\ell - 1, L)$  and 0, respectively.

Using both the relative frontier  $\rho$  and the relative position  $(i, s)$ ,  $\mathcal{A}'$  can ensure that entering a cell for the first time, may be done only once all the information, which is required to determine the symbol to write on the cell, has been gathered. More precisely, when  $\mathcal{A}'$  moves its head to the frontier cell (Line 9), it stores a pair  $(p, q)$  in its finite control, such that:

- $p \in Q$  is the state entered by  $\mathcal{A}$  in the simulated computation, when visiting for the first time the corresponding cell;

- $q \in Q_\perp$  is either  $\perp$  or a state such that  $(\nu(\rho), q) \in \tau_{zx}$ , where  $zx$  denotes the content to the left of the right block of the window (*i.e.*, the block of the frontier cell) in the simulated computation.

The states  $p$  and  $q$  are stored in two variables, respectively named **frontierState** and **tableState**, which are updated through two calls to the subroutine **simulateLeft**:

- from one cell to the left of the frontier, to update **frontierState** (Line 4);
- from the last cell of the preceding block, to update **tableState** (Line 7).

Once  $\mathcal{A}'$  has updated the variables **frontierState** and **tableState**, it moves the head to the cell at relative position  $(\rho, R)$  (Line 9), and reads the input symbol  $\sigma \in \Sigma \cup \{\triangleleft\}$  (Line 10). If  $\sigma = \triangleleft$  then  $\mathcal{A}'$  enters a final mode in which it calls the subroutine **simulateLeft** with argument **frontierState** from the current position, and accepts, after violating the endmarker, if the updated value of **frontierState** is a final state of  $\mathcal{A}$  and rejects otherwise (Lines 16 and 17). If  $\sigma \neq \triangleleft$  then  $\mathcal{A}'$  simulates a stationary overwriting transition of  $\mathcal{A}$  (Lines 11 to 15). Formally, it selects a transition  $(p', \gamma, 0) \in \delta(p, \sigma)$ , where  $p$  is the state stored in **frontierState**, updates the variable **frontierState** with  $p'$ , and overwrites the cell content with  $(\gamma, h)$  where  $h \in T$  is defined as follows according to the value  $q \in Q_\perp$  of **tableState**. If  $T = \{0, 1\}$  then  $h = 1$  if  $\mu(\rho) = (\nu(\rho), q)$  and  $h = 0$  otherwise. If  $T = Q_\perp$  then  $h = q$ . It then repeats the procedure with the updated relative frontier. In the case  $\rho = \ell - 1$ , the window is shifted to the right, in such a way that the head is positioned on the rightmost cell of its left block. This is formally done by setting the relative position to  $(\ell - 1, L)$  and the relative frontier to 0.

### 3.2.3. Implementation details for the auxiliary operations and procedures

*The operation selectTransition*

Our subroutines **simulateLeft** and **main** use a basic operation named **selectTransition** (Lines 11 and 22). This operation takes a state  $p \in Q$  and a symbol  $\sigma \in \Gamma$  as arguments, and returns a tuple  $(q, \gamma, d) \in \delta(p, \sigma)$ . When no such transition exists, it returns  $(\perp, \sigma, 0)$ . Notice that the operation is nondeterministic only



if  $\mathcal{A}$  is nondeterministic.

*The operation **write***

In our simulation,  $\mathcal{A}'$  overwrites symbols in  $\Sigma$  with symbols in  $(\Gamma \setminus \Sigma) \times T$  by performing an operation named **write**. This operation takes two arguments,  $\gamma \in \Gamma \setminus \Sigma$  and  $r \in Q_\perp$ . When  $T = \{0, 1\}$ , it compares  $r$  to the state  $q$  such that  $\mu(i) = (\nu(i), q)$  where  $i$  is the index of the current relative position. If  $r = q$  then the symbol  $(\gamma, 1)$  is written, otherwise (including the case  $r = \perp$ ) the symbol  $(\gamma, 0)$  is written. When  $T = Q_\perp$ , the routine simply overwrites the content of the currently scanned cell with  $(\gamma, r)$ .

*The subroutine **readFromTable***

This subroutine was introduced in Section 3.2.1. It is used to prevent the head of  $\mathcal{A}'$  to move to the portion of the tape on the left of the current window. It is always called from the leftmost position of the window. In particular, this position is the first one of a frozen block  $\tilde{x} \in (\Gamma' \setminus \Sigma)^\ell$ , supposed to contain on its second track the encoding of the table  $\tau$  that describes the possible computation segments to the left of the window. The routine takes a global variable **var** as argument, initially containing a state  $p \in Q$ .

The procedure operates in two modes. First, it moves the head rightward until reaching a position  $i$  of  $\tilde{x}$  such that  $\nu(i) = p$ . Second, it moves the head backward to the first position of  $\tilde{x}$  and halts. When switching from the former to the latter mode at position  $i$ , the variable **var** is updated with an element  $q \in Q_\perp$ , which is deduced from the scanned symbol  $(\gamma, t) \in (\Gamma \setminus \Sigma) \times T$ , and the current relative position  $i$ , as now explained. If  $T = \{0, 1\}$  then, according to whether  $t$  equals 0 or 1,  $q$  is equal to  $\perp$  or is the state such that  $\mu(i) = (\nu(i), q)$ , respectively. If  $T = Q_\perp$  then  $q$  is equal to  $t$ . In the nondeterministic case, such a position  $i$  is nondeterministically chosen. In the deterministic case, however,  $\mathcal{A}'$  can select the position  $i$  deterministically. Indeed, when  $T = Q_\perp$ , there exists exactly one  $i$  such that  $\nu(i) = q$ . On the other hand, when  $T = \{0, 1\}$ , several such indices may exist, but at most one is such that the symbol  $(\gamma, t)$  written at the corresponding position satisfies  $t = 1$ , by determinism of  $\mathcal{A}$ . In this latter case,

when no such  $i$  exist, the procedure sets the variable `var` to  $\perp$ . Thus, the routine deterministically finds this position, and returns the image  $q \in Q_\perp$  of  $p$  by the *functional* table  $\tau$  written on  $\tilde{x}$ .

**Lemma 2.** *The procedure `readFromTable` can be implemented using 2 states, not counting the global variables `var` and `relativePosition`. Furthermore, the implementation is deterministic whenever  $\mathcal{A}$  is deterministic.*

*Proof.* In all the cases described above, the procedure needs only one state to move the head rightward until finding the correct information, and a second state to move the head back to the initial position, namely to relative position  $(0, L)$ . The recovered information is directly stored in `var`.  $\square$

#### *The subroutine `simulateLeft`*

This subroutine was introduced in Section 3.2.1. It is used to update the variables `frontierState` and `tableState` before visiting the frontier cell. Hence, the routine has two call-modes:

- one for updating `frontierState` starting from one cell to the left of the frontier;
- the other one for updating `tableState` starting from the rightmost cell of the left block of the window.

Let us denote by `var` the variable to be updated and by  $(i, s)$  the relative position from which the routine is called. Let  $zxy \in \triangleright(\Gamma \setminus \Sigma)^*$  be the projection on  $(\Gamma \setminus \Sigma) \cup \{\triangleright\}$  of the tape content up to the starting position, with  $x$  corresponding to the left block of the current window. During the computation, `simulateLeft` has access to the content of the window up to position  $(i, s)$ . It basically performs a direct simulation of  $\mathcal{A}$  on the corresponding part of the tape, and uses the procedure `readFromTable` in order to simulate computations that occur to the left of the window, as explained above. Moreover, if a rightward transition  $(q, \gamma, +1) \in \delta(r, \gamma)$  from the last position of  $zxy$  has to be simulated, then the procedure halts without performing the right move, namely

---

**Procedure 2: simulateLeft(var)**

---

```

/* the variables relativePosition and relativeFrontier are not indicated
   and are supposed to be automatically updated */
Input: a variable var in read/write mode, initially containing a state in  $Q$ 
Output: halts with var containing a state or the special symbol  $\perp$ 
18 let  $(i, s) = \text{relativePosition}$ 
19 clock  $\leftarrow 2\ell n$ 
20 while var  $\neq \perp$  and clock  $> 0$  do
21   let  $\gamma \in \Gamma_{\triangleright \triangleleft} \setminus \Sigma$  be the first track symbol of the currently scanned cell
22   let  $(q, \gamma, d) = \text{selectTransition}(\text{var}, \gamma)$ 
23   var  $\leftarrow q$ 
24   if var  $= \perp$  then
25     break
26   else if relativePosition  $= (i, s)$  and  $d = +1$  then
27     break
28   else if relativePosition  $= (0, L)$  and  $d = -1$  then
29     readFromTable(var)
30   else
31     move head according to  $d$ 
32     clock  $\leftarrow \text{clock} - 1$ 
33 if clock  $= 0$  then var  $\leftarrow \perp$ 
34 move head rightward until reaching position  $(i, s)$ 

```

---

at relative position  $(i, s)$ , and updates the variable `var` to the value  $q$ . Notice that the direct simulation is deterministic if  $\mathcal{A}$  is deterministic, since the procedure `readFromTable` is deterministic in this case, by Lemma 2.

However, this naive approach might fail because the direct simulation may enter loops and never halt. In order to handle this issue, we need to detect computational loops. We proceed as follows. If  $(p, q) \in \tau_{zxy}$  then this can be witnessed by a direct simulation which never repeats a configuration. In particular, the same state cannot be entered twice at the same position. Since the procedure `simulateLeft` operates in a read-only window of size at most  $2\ell$ , any repetition-free computation has length bounded by  $2\ell n$  (counting the calls to the subroutine `readFromTable` as a single move). Hence, by using a clock of size  $2\ell n$ , stored in the finite control of the simulating device, we can enforce the procedure to halt. Only runs that halted before this time limit may return a state while “killed” runs will return  $\perp$ . Notice that the clock yields a polynomial increase of the size of the simulating machine only.

Let us focus on the implementation details of `simulateLeft` given in Procedure 2. The subroutine starts by storing the value  $(i, s)$  of the current relative position (Line 18) and the clock to  $2\ell n$  (Line 19). After that,  $\mathcal{A}'$  simulates  $\mathcal{A}$  by reading the first track of the scanned cell and using the transition function of the simulated machine (Lines 21 and 22). The next simulated state reached by  $\mathcal{A}$  is stored in the variable `var` (Line 23). If, by incompleteness of the transition function of  $\mathcal{A}$  no such state exists, `var` gets value  $\perp$  and the procedure ends after moving its head rightward to the position  $(i, s)$  from which it was called (Lines 24, 25 and 34). In case a right move from relative position  $(i, s)$  is detected, the procedure ends without moving rightward (Lines 26 and 27). If a left move from the relative position  $(0, L)$  is detected,  $\mathcal{A}'$  calls `readFromTable` in order to simulate the computation segment to the left of the window (Lines 28 and 29). Otherwise,  $\mathcal{A}'$  can directly simulate the transition performed by  $\mathcal{A}$ : the simulating machine moves its head according to the simulated transition (Line 31). After each simulated move<sup>2</sup> of  $\mathcal{A}$ , the value of the clock is decremented (Line 32). If after  $2\ell n$  simulated moves the routine has not halted, then it updates the value of `var` with  $\perp$ , moves the head rightward until reaching the relative position  $(i, s)$ , and halts (Lines 33 and 34).

Hence, implementing `simulateLeft` requires only a polynomial number of states in  $n$ .

**Lemma 3.** *The subroutine `simulateLeft` can be implemented using  $12\ell n + 2$  states, not counting the global variables `var`, `relativePosition`, and `relativeFrontier`. Furthermore, it is deterministic if  $\mathcal{A}$  is deterministic.*

*Proof.* Using Lemma 2, the implementation of `simulateLeft` given in Procedure 2 is deterministic when  $\mathcal{A}$  is deterministic. Furthermore, it uses the following state components.

- A binary state component is required for keeping track of the call-mode

---

<sup>2</sup>Here, we consider the simulated computation segments to the left of the window, which are recovered through call to `readFromTable` (Line 29), as single moves.

in which the procedure is operating. Moreover, since the mode also determines the position from which the procedure is called, and because the relative frontier (stored in the variable `relativeFrontier`) is not modified during the execution of the procedure, no further state components are necessary for storing the initial value  $(i, s)$  of `relativePosition`.

- A state component of size  $2\ell n$  is required for storing the value of the clock (we do not need to store the value 0, as the device can directly enter a failure state when decrementing the clock from value 1).
- A state component of size 3 storing the three internal modes of the procedure, namely the main mode and the two sub-modes resulting from the calls to the routine `readFromTable` (Line 29) by Lemma 2.
- One failure state for each call-mode is required for moving the head rightward until reaching the initial position  $(i, s)$  when the procedure fails (Line 34).

Hence, the total number of states required for implementing `simulateLeft` is  $2 \cdot (2\ell n \cdot 3 + 1)$ , not counting the global variables `var`, `relativePosition`, and `relativeFrontier`.  $\square$

#### 3.2.4. Sipser's simulation

When the simulated 1-LA  $\mathcal{A}$  is deterministic it is possible to use a finer implementation of `simulateLeft`, thus avoiding the size-expensive clock. This finer implementation is an adaptation of a construction due to Sipser, that avoids deterministic loops in deterministic Turing machines by a clever backward simulation [15]. A version for 2DFAs has been presented in [16], where it is shown that a linear increase of the size is sufficient for simulating any 2DFA with a halting equivalent one. We first recall the main ideas of this simulation and then we show how it can be adapted for implementing the procedure `simulateLeft`.

Let  $\mathcal{B}$  be a 2DFA. Without loss of generality (see [16, Lemma 3.1]), we can suppose that  $\mathcal{B}$  cannot perform stationary moves, that it has exactly one final state  $q_F$ , that acceptance is made by entering  $q_F$  at the leftmost position, namely on the left endmarker, and that furthermore no transition can be performed from that point, *i.e.*, from state  $q_F$  reading  $\triangleright$ . Then, given an input word  $w$ , we consider the *configuration graph*  $\mathcal{G}$  of  $\mathcal{B}$  on  $w$ , namely the directed graph such that the vertices are the configurations of  $\mathcal{B}$  on  $w$  and there is an edge from  $c$  to  $c'$  if  $c \vdash c'$ . In particular,  $w$  is accepted by  $\mathcal{B}$  if and only if there exists a path in  $\mathcal{G}$  from the initial configuration  $c_I$  to the unique final configuration  $c_F$ . Let us focus on the connected component of  $c_F$ . Since  $\mathcal{B}$  is deterministic and because  $c_F$  has no successor by assumption, this component is a tree rooted in  $c_F$ . Moreover,  $w$  is accepted if and only if  $c_I$  occurs in this tree. Hence, by performing a depth-first-search in the tree, one can check whether  $c_I$  is in the component, and thus decide whether the word  $w$  is accepted. This depth-first-search idea can be implemented deterministically, starting from  $c_F$ , using four copies of each state only. This yields a  $(4 \cdot \#Q_{\mathcal{B}})$ -state halting 2DFA which is equivalent to  $\mathcal{B}$ , where  $Q_{\mathcal{B}}$  denotes the state set of  $\mathcal{B}$  [16]. It should be noticed that when  $c_I$  does not belong to the tree, the simulating halting 2DFA halts in a configuration that matches the root  $c_F$  after having tried all its subtrees.

We shall adapt this construction to our case, for implementing the procedure **simulateLeft**. Notice that the direct simulation of  $\mathcal{A}$  on the corresponding frozen portion of the tape is a read-only deterministic computation. Remember that **simulateLeft** can operate in two call-modes. Let us fix one of these two modes. We denote by **var** the variable to update, by  $p$  its initial content, by  $(i, s)$  the starting relative position, and by  $\rho$  the relative frontier. Let  $zwX$  be the content of the tape to the left of the starting position, with  $z$  being the portion to the left of the window and  $|X| = 1$ .

We consider a 2DFA variant  $\mathcal{C}$  simulating  $\mathcal{A}$ , which starts from and ends in some specified position. We assume that it has access to the variables **relativePosition** and **relativeFrontier**. It basically performs an unclocked version of

Procedure 2 in which the failure state (Line 34) has been thrown away. Furthermore, we ignore the last update of `var`, when the image  $q$  of  $p$  by  $\tau_{zwX}$  is found. We can suppose that  $\mathcal{C}$  never performs stationary moves. Indeed, on the one hand,  $\mathcal{A}$  does not perform any stationary move when working on frozen symbols by assumption. On the other hand, without increasing the size of the device, we can eliminate the stationary moves possibly resulting from the calls to the subroutine `readFromTable`, by using classical techniques. Moreover,  $\mathcal{C}$  can be implemented using 3 states only, not counting the variable `var`, according to Lemma 2 (one state for the main mode, and two additional states for the calls to the sub-routine `readFromTable`). Remember that  $X$  is the symbol written at relative position  $(i, s)$  and let  $R(X) = \{r \mid \exists q, \delta(r, X) = \{(q, X, +1)\}\}$ . An *accepting configuration* of  $\mathcal{C}$ , is a configuration in which the head is positioned at relative position  $(i, s)$  (thus scanning  $X$ ), the internal state corresponds to its main mode (*i.e.*, not to a call to `readFromTable`), and `var` contains a state  $r \in R(X)$ .

Observe that, although we dropped the last update of `var` with respect to Procedure 2, it is possible to recover it from the halting point of  $\mathcal{C}$ . Indeed, this value is either  $q \in Q$  if  $\mathcal{C}$  halted in an accepting configuration with `var` storing  $r \in R(X)$  such that  $\delta(r, X) = \{(q, X, +1)\}$ , or  $\perp$  otherwise. Notice that  $\mathcal{C}$  does not use the endmarkers but rather the relative position, to ensure that the head stays between  $(0, L)$  and  $(i, s)$ . In particular, the *initial configuration*  $c_I$  of  $\mathcal{C}$  is at relative position  $(i, s)$  with the state corresponding to its main mode and `var` storing  $p$ .

The main issue for adapting Sipser's construction to our case, is that the target configuration  $c_F$  is initially unknown. Indeed, it is the role of `simulateLeft` to find the image  $q$  of  $p$  by  $\tau_{zwX}$  (when defined). In order to solve this issue, we apply the Sipser simulation of  $\mathcal{C}$  for each value  $r \in R(X)$  of `var`. By taking such  $r$ 's in order, we only need 4 internal sub-modes for the simulation, as in [16]. Indeed, if for some target configuration  $c_F(r)$  the simulation does not find  $c_I$ , then it halts in a configuration that encodes  $c_F(r)$ . In particular, `var` contains  $r$  and can thus be updated with the next value from  $R(X)$ . If no suc-

cessor of  $r$  exists, then  $\mathcal{C}$  can never reach an accepting configuration, namely  $c_I$  is in none of the trees rooted in the  $c_F(r)$ 's. Thus, our procedure updates `var` with the symbol  $\perp$  and halts. If otherwise the configuration  $c_I$  has been found during a simulation starting from some  $c_F(r)$ , then the direct execution of  $\mathcal{C}$  halts in  $c_F(r)$ . Thus, it is enough to directly simulate  $\mathcal{C}$  from  $c_I$  and to update `var` according to the state  $r$  which is recovered when the execution halts. More precisely, once  $\mathcal{C}$  has reached  $c_F(r)$ , our procedure reads the symbol  $X$  and updates the variable `var` which contains  $r \in R(X)$  with the state  $q$  such that  $\delta(r, X) = \{(q, X, +1)\}$ .

We now evaluate the size of this deterministic version of `simulateLeft`, which works independently of the chosen table encoding (*i.e.*, for any  $T, \ell$ ), so long as  $\mathcal{A}$  is deterministic.

**Lemma 4.** *If  $\mathcal{A}$  is deterministic, then, independently of the encoding of the Shepherdson tables on the tape, the procedure `simulateLeft` can be implemented deterministically using 30 internal states, not counting the global variables `var`, `relativePosition`, and `relativeFrontier`.*

*Proof.* We evaluate the implementation described above. As in the implementation given in Section 3.2.3, we need a binary component for storing the call-mode of the procedure. This component is sufficient to recover the starting position  $(i, s)$ , possibly by reading the value of `relativeFrontier` which is not modified during the execution of the procedure.

Then the 3-mode automaton  $\mathcal{C}$  given above is simulated a first time, by Sipser's construction, using 4 sub-modes. If the simulation succeeds then  $\mathcal{C}$  is simulated a second time in order to recover the value to store in `var`. Otherwise, no further state is needed, as a failure necessarily occurs at position  $(i, s)$  from which `var` is updated to  $\perp$  without moving the head.

Thus, not counting the global variables `relativePosition`, `relativeFrontier`, `frontierState`, and `tableState`, we obtain that  $2(12 + 3)$  states are enough.  $\square$



### 3.3. Properties of $\mathcal{A}'$

In this section, we state the main properties of the 1-LA  $\mathcal{A}'$  that has been obtained from  $\mathcal{A}$  by the simulation defined in Section 3.2. Notice that several simulations have been described. We differentiate them only when required.

**Lemma 5.**  *$\mathcal{A}'$  is equivalent to  $\mathcal{A}$ .*

*Proof.* We first prove that  $L(\mathcal{A}) \subseteq L(\mathcal{A}')$ . Consider an accepting computation  $\mathbf{c} = c_0, c_1, \dots, c_t$  of  $\mathcal{A}$  on some input  $w = w_1 \cdots w_m \in \Sigma^*$ , where  $c_0 = q_0 \triangleright w \triangleleft$  and  $c_t = \triangleright x \triangleleft \bullet q_F$ , for some  $q_F \in F$  and  $x = x_1 \cdots x_m \in (\Gamma \setminus \Sigma)^*$ . We can extract from  $\mathbf{c}$  the sequence  $c_{j_1}, \dots, c_{j_m}$  of configurations in which the head is scanning a symbol of  $\Sigma$ , namely, for each  $i \in \{1, \dots, m\}$ ,  $c_{j_i} = \triangleright x_1 \cdots x_{i-1} \bullet p_i \bullet w_i \cdots w_m \triangleleft$  for some state  $p_i$ . Notice that  $p_i$  is necessarily the first state entered at position  $i$  in  $\mathbf{c}$ . Moreover, since by assumption  $\mathcal{A}$  has the form given in Lemma 1, we have  $c_{j_{i+1}} = \triangleright x_1 \cdots x_{i-1} \bullet p'_i \bullet x_i w_{i+1} \cdots w_m \triangleleft$  for some state  $p'_i$ . In particular,  $(p'_i, x_i, 0) \in \delta(p_i, w_i)$  is the stationary transition which is performed when overwriting the content of the cell at position  $i$  during the computation  $\mathbf{c}$ . For convenience, we define  $c_{j_{m+1}}$  to be the first configuration of the form  $\triangleright x_1 \cdots x_m \bullet p_{m+1} \bullet \triangleleft$  for some state  $p_{m+1}$ . We also set  $x_0 = \triangleright$ ,  $x_{m+1} = \triangleleft$ ,  $p_0 = p'_0 = q_0$ ,  $p'_{m+1} = p_{m+1}$ , and  $p_{m+2} = q_F$ . For each  $i \in \{0, \dots, m+1\}$ , we have  $(p'_i, p_{i+1}) \in \tau_{x_0 \cdots x_i}$ .

The simulating 1-LA  $\mathcal{A}'$  recovers  $\mathbf{c}$  by successively storing the states  $p'_0, p_1, p'_1, \dots, p_{m+1}, p'_{m+1}, p_{m+2}$  in its internal variable `frontierState`, while visiting the corresponding cells from left to right. More precisely, for each  $i \in \{0, \dots, m+2\}$ , when  $\mathcal{A}'$  enters the  $i$ -th tape cell for the first time, the variable `frontierState` contains the state  $p_i$ . It is routine to show by induction that, at each iteration of the while loop in Procedure 1 (Lines 3 to 15), as soon as the left block of the current window encodes the correct table, the two calls to the subroutine `simulateLeft` can recover the right information, namely `frontierState` is updated from  $p'_i$  to  $p_{i+1}$  on Line 4, and `tableState` is updated from  $\nu(\rho)$  to  $q$  on Line 7, where  $q$  is such that:

- if  $T = \{0, 1\}$  then either  $q$  is a state  $r$  such that  $(\nu(\rho), r) \in \tau_{x_0 \cdots x_k}$ , or  $q = \perp$ ;

- if  $T = Q_\perp$  then  $q$  is the image of  $\nu(\rho)$  by  $\tau_{x_0 \dots x_k}$  if defined, or  $\perp$  otherwise;

where  $k$  is the rightmost position of the left block of the window.

Conversely, updating `frontierState` in  $\mathcal{A}'$  is done only by performing direct simulation of  $\mathcal{A}$  that may read some table  $\tau$ , which have previously be written on the frozen content of the tape. By induction on the frontier position, we can prove that  $\tau$  is a relation included in  $\tau_{x_0 \dots x_{k-\ell}}$ , where  $k$  is the rightmost position of the left block of the window. (We further have  $\tau = \tau_{x_0 \dots x_{k-\ell}}$  when  $\mathcal{A}$  is deterministic.) Hence, every state recovered through `readFromTable` and `simulateLeft`, correspond to a state that can be entered by  $\mathcal{A}$  from the corresponding configuration at the corresponding position. Thus, for each accepting computation of  $\mathcal{A}'$ , one can find a simulated accepting computation of  $\mathcal{A}$ , whence  $L(\mathcal{A}') \subseteq L(\mathcal{A})$ .  $\square$

We have shown how  $\mathcal{A}'$  simulates  $\mathcal{A}$  in a halting manner, by shifting a virtual window to the right during its computation, and by restricting local head moves to the current window of size  $2\ell$ . We now evaluate the size of  $\mathcal{A}'$ . We point out that, as long as  $\mathcal{A}$  is deterministic, the two possible encodings of the Shepherdson tables, namely using  $T = \{0, 1\}$  and  $\ell = n^2$  or using  $T = Q_\perp$  and  $\ell = n$ , are possible. For both, the Sipser simulation yields a smaller size increase with respect to the clock trick used for the general case. Though in the deterministic case, the smallest simulating 1-LA is obtained by combining the second encoding with Sipser simulation, it should be noticed that using the first encoding yields a smaller working alphabet, whose size does not depend on  $n$ .

**Lemma 6.**  *$\mathcal{A}'$  has polynomial size with respect to  $\mathcal{A}$ . More precisely, we obtain the following simulation costs:*

<i>case</i>	<i>technique</i>	<i>states</i>	<i>working symbols</i>
<i>nondeterministic</i>	<i>clock</i> / $\ell = n^2$	$O(n^9)$	$2 \cdot \#(\Gamma \setminus \Sigma)$
<i>deterministic</i>	<i>Sipser</i> / $\ell = n^2$	$O(n^6)$	$2 \cdot \#(\Gamma \setminus \Sigma)$
	<i>Sipser</i> / $\ell = n$	$O(n^4)$	$(n + 1) \cdot \#(\Gamma \setminus \Sigma)$

*Proof.* The set of working symbols of  $\mathcal{A}'$  is  $\Gamma' \setminus \Sigma = (\Gamma \setminus \Sigma) \times T$ . In both non-deterministic and deterministic cases, the finite control uses several components:

- the variable `relativeFrontier` of size  $\ell$ ;
- the variable `relativePosition` of size  $2\ell - 1$  (the value  $(\ell - 1, R)$  is never used, since the relative position is always to the left of the relative frontier);
- the variable `frontierState` of size  $n$  (the value  $\perp$  is unnecessary, since updating `frontierState` with  $\perp$  implies rejecting the input);
- the variable `tableState` of size  $n + 1$ ;
- the state components used to implement `simulateLeft` of size:
  - $12\ell n + 2$  using the clock (general case, see Section 3.2.3);
  - 30 using Sipser's construction (deterministic case only, see Section 3.2.4).

In both cases, the size includes the two sub-modes used in the implementation of the routine `readFromTable`, cf. Lemma 2.

□

Let us now evaluate the time used by the simulating machine  $\mathcal{A}'$ .

**Lemma 7.** *In every computation of  $\mathcal{A}'$ , each tape cell is visited a number of times which is bounded by some polynomial in the size of  $\mathcal{A}$ .*

*Proof.* Let us fix a cell  $c$ . As  $\mathcal{A}'$  is loop-free, each time the head visits  $c$  it must have a different state or a different tape content. A tape modification between two visits of  $c$  is restricted to cells from the right block of the current window containing  $c$ . The number of successive tape modifications in a window is bounded by  $\ell$ . Indeed, after  $\ell$  overwritings the window is shifted. The cell  $c$  may occur in two successive windows: first in the right part and, after shifting the window, in the left part. Thus, the number of visits to the cell  $c$  is bounded by  $2\ell n'$ , where  $n'$  is the number of states of  $\mathcal{A}'$ , which is polynomial in the number of states of  $\mathcal{A}$  as seen in Lemma 6. The number of visits to each cell is hence bounded by a polynomial in the size of  $\mathcal{A}$ . □

As a consequence,  $\mathcal{A}'$  operates in linear time with respect to the input length.

We can observe that, by the use of the state components `relativePosition` and `relativeFrontier`, our simulating 1-LA always “knows” where the frontier is. Roughly, this means that  $\mathcal{A}'$  does not use the meta-instruction “move rightward until finding the leftmost cell that has not been visited so far” which was used by the 1-LA described in Example 1.

**Lemma 8.**  *$\mathcal{A}'$  “knows” where the frontier is, namely, there exist special states that are entered exactly when visiting a tape cell for the first time.*

*Proof.* In Procedure 1, when entering the frontier cell (Line 9), the simulating device enters a particular mode from which the cell is scanned (Line 10) in order to simulate a stationary overwriting transition of  $\mathcal{A}$  (Lines 11 to 15). Hence, we can exhibit the set of states corresponding to this special mode.  $\square$

#### 4. Main Result and Consequences

We are now able to state our results as consequences of the properties of  $\mathcal{A}'$  stated in Section 3.3. See Figure 1 for a summary of these results.

##### 4.1. Main Result: Conversion into Linear-Time 1-Limited Automata

Our main result shows that operating in super-linear time is not essential for 1-LAS, if allowing a polynomial increase in the number of states.

**Theorem 1.** *Each 1-LA (resp. deterministic 1-LA) admits an equivalent linear-time 1-LA (resp. deterministic 1-LA) of polynomially larger size.*

*Proof.* We start with a 1-LA. By paying a linear increase of its size and preserving determinism, we transform it into an equivalent 1-LA  $\mathcal{A}$  which performs stationary moves exactly when rewriting a cell content, by Lemma 1. Then we apply the above construction in order to obtain the 1-LA  $\mathcal{A}'$  equivalent to  $\mathcal{A}$ , by Lemma 5. If  $\mathcal{A}$  is deterministic then so is  $\mathcal{A}'$ . By Lemma 6, the size of  $\mathcal{A}'$  is bounded by some polynomial in the size of  $\mathcal{A}$ . By Lemma 7,  $\mathcal{A}'$  operates in linear time in the length of the input.  $\square$

#### 4.2. Conversion into Weight-Reducing Hennie machines

Linear-time 1-LAS are particular cases of Hennie machines (*i.e.*, linear-time linear bounded automata), hence, it follows from the above result that any 1-LA can be transformed into a Hennie machine with a polynomial increase of the size only. Using Lemma 7, we can actually obtain the stronger result that the 1-LA can be transformed into a *weight-reducing Hennie machine* of polynomial size. Informally, weight-reducing Hennie machines are Hennie machines in which each overwriting is decreasing with respect to some fixed order on the working alphabet. As a consequence, after overwriting a cell with a minimal symbol, such a machine cannot visit the cell again. See [6] for formal definition and study of the model.

**Theorem 2.** *Each 1-LA (resp. deterministic 1-LA) admits an equivalent weight-reducing Hennie machine (resp. deterministic weight-reducing Hennie machine) of polynomial size.*

*Proof.* Following [6, Lemma 4], it is enough to modify the 1-LA  $\mathcal{A}'$  obtained by the above construction, in such a way that each time a frozen cell is visited, it is overwritten with a copy of the frozen symbol, that encodes the number of visits to the cell. Since, by Lemma 7, the total number of visits of a cell in a computation of  $\mathcal{A}'$  is bounded by some polynomial in the size of  $\mathcal{A}$ , the transformation yields an equivalent weight-reducing Hennie machine which has polynomial size with respect to the simulated 1-LA. Furthermore, the conversion clearly preserves determinism.  $\square$

#### 4.3. Conversion into Two-Way Automata with Common Guess

Some 1-LAS have a particular behavior, which can be decomposed into two phases. In the first phase, they nondeterministically rewrite the content of the whole tape during a left-to-right traversal of the input. Then, in the second phase, they perform a two-way read-only computation over the overwritten tape. To formally define this kind of 1-LAS, we introduce the following model.

**Definition 3.** A 2NFA (resp. 2DFA) with common guess (2NFA+cg, resp. 2DFA+cg)<sup>3</sup> is a tuple  $\langle \mathcal{A}, \Sigma, \Delta \rangle$  where  $\Sigma$  and  $\Delta$  are two alphabets and  $\mathcal{A}$  is a 2NFA (resp. 2DFA) over the product alphabet  $\Sigma \times \Delta$ .

The model is aimed to recognize languages over  $\Sigma$ . Its dynamics is defined as for two-way automata, but a nondeterministic pre-computation initially marks each input symbol with a symbol from  $\Delta$ .<sup>4</sup> Hence, the read-only automaton  $\mathcal{A}$  has access to both the input symbol and the guessed additional information. The language accepted, denoted  $L(\langle \mathcal{A}, \Sigma, \Delta \rangle)$ , is defined as the projection, denoted  $\pi_1$ , of  $L(\mathcal{A})$  on the alphabet  $\Sigma$ , i.e.,  $L(\langle \mathcal{A}, \Sigma, \Delta \rangle) = \pi_1(L(\mathcal{A}))$ . In other terms, a word is accepted by  $\langle \mathcal{A}, \Sigma, \Delta \rangle$  if for some guess, the enriched word in  $(\Sigma \times \Delta)^*$  is accepted by  $\mathcal{A}$ . We point out that, due to the common guess, 2DFA+cg's are nondeterministic devices.

Let us detail the connection between 1-LAS and 2FA+cg's. It is easy to turn a 2FA+cg into an equivalent nondeterministic 1-LA of the same size, by simply guessing and writing the symbols from  $\Delta$  when visiting the cells for the first time. It is however *a priori* not clear whether a converse transformation with reasonable size cost exists. The main issue for such a conversion is that, at any time during a computation of a 1-LA, a position of the tape is identified as being the leftmost cell that has not been visited so far, namely the current frontier. In particular, a 1-LA can use meta-instructions making use of this identified position, such as “move the head rightward to the frontier cell”, as it is the case in Example 1. Nevertheless, when a 1-LA does not use such kind of instructions, that is, if it always “knows” when it enters a cell for the first time (before scanning its content), then it is easy to convert it to an equivalent 2FA+cg of similar size. Formally, the property of always “knowing” where the

---

<sup>3</sup>2DFA+cg's also correspond to *synchronous two-way deterministic finite verifiers* [17].

<sup>4</sup>Though the model is motivated by special behaviors of 1-LAS whence the nondeterministic pre-computation is naturally thought as being one-way left-to-right, there is no reason to impose this. Here, the marking is uniform meaning that each cell is independently marked by a nondeterministically-chosen symbol of  $\Delta$ . This operation is connected with existential set quantification in *monadic second order logic*, see, e.g., [13].

frontier is can be expressed by specifying the states that are entered exactly when visiting a tape cell for the first time. In order to simulate such a 1-LA (with input alphabet  $\Sigma$  and working alphabet  $\Gamma$ ) with a 2FA+cg (with common guess alphabet  $\Delta$ ), it is indeed enough to first guess the working symbols that will be written on the tape at the end of the computation (thus, setting  $\Delta = \Gamma$ ), and then simulate the 1-LA in a read-only manner, using the symbol component in  $\Sigma$  when the cell is visited for the first time (which is determined by the current state) or the symbol component in  $\Gamma$  otherwise, while checking that the guessed symbols correspond to the symbol overwritten during the simulated computation. Notice that, so obtained, the resulting device is a 2DFA+cg (resp. is halting) when the source 1-LA is deterministic (resp. halting).

Concerning the conversion of arbitrary 1-LAS (*i.e.*, not “knowing” where the frontier is) into equivalent 2FA+cgs, it is a non-trivial consequence of our main construction that with a polynomial increase of the size only, this can be achieved.

**Theorem 3.** *Each 1-LA (resp. deterministic 1-LA) admits an equivalent halting 2NFA+cg (resp. 2DFA+cg) of polynomial size.*

*Proof.* By Lemma 8,  $\mathcal{A}'$  “knows where the frontier is”. Hence, by applying the above-given conversion, we can obtain an equivalent 2FA+cg of polynomial size, which is halting by Lemma 7. Furthermore, if  $\mathcal{A}'$  is deterministic, then the resulting device is a 2DFA+cg.  $\square$

In the nondeterministic case, this last result is of particular interest. Indeed, 2NFA+cg’s can be seen as particular cases of 1-LAS. (It is not the case for 2DFA+cg’s with respect to deterministic 1-LAS.) Hence, Theorem 3 gives a kind of normal form for nondeterministic 1-LAS. In particular, it is easy to modify such a 1-LA in order to recognize the reverse of its accepted language.

**Corollary 1.** *Each 1-LA  $\mathcal{A}$  can be transformed into a nondeterministic 1-LA  $\mathcal{A}'$  with a polynomial increase of the size, such that  $L(\mathcal{A}') = L(\mathcal{A})^R$ .*

*Proof.* Given a 1-LA  $\mathcal{A}$ , we can obtain an equivalent 2NFA+cg by Theorem 3. By replacing left move by right move and *vice versa* on each transition of its underlying automaton, we can obtain a 2NFA+cg of same size, which recognizes  $L(\mathcal{A})^R$ . This 2NFA+cg can in turn be viewed as a nondeterministic 1-LA.  $\square$

#### 4.4. Lower bounds

Concerning the size cost of the simulation of 2DFA+cg by deterministic 1-LA, using the language  $L_n$  from Example 1, we can prove an exponential gap in the deterministic case.

**Theorem 4.** *Let  $L_n$  be the language of Example 1. Hence*

$$L_n^R = \{x_k x_{k-1} \cdots x_0 \mid k \in \mathbb{N}, \text{ for each } i: x_i \in \{a, b\}^n, \text{ for some } j \neq 0: x_j = x_0\}.$$

*Then,*

1.  $L_n^R$  is accepted by a 2DFA+cg, a linear-time nondeterministic 1-LA, or a deterministic weight-reducing Hennie machine of size polynomial in  $n$ ;
2. any 1DFA recognizing  $L_n^R$  requires  $2^{2^n}$  states;
3. any deterministic 1-LA recognizing  $L_n^R$  requires  $O(2^n)$  states.

*Proof.* Example 1 describes a deterministic 1-LA recognizing  $L_n$ , whose size is linear in  $n$ . By applying Theorems 2 and 3, we respectively obtain equivalent deterministic weight-reducing Hennie machine and 2DFA+cg of polynomial size. Both models can be transformed with at most a linear increase in size, in order to accept the reverse of the language, thus proving Item 1. For both models, it is indeed enough to initially move the head to the right endmarker, and then simulate the two-way device in opposite direction, that is, replacing left moves of the head by right ones and *vice versa*. In the case of 2DFA+cg this yields a constant increase of the size of the model (only one state should be added for the initial mode). In the case of weight-reducing Hennie machines, since during the initial traversal of the input the cells should be overwritten in a decreasing way



(in order to preserve the weight-reducingness property), we should in addition add a fresh copy of each input symbol to the set of working symbols.

Using a simple distinguishability argument, we can prove Item 2. Finally, Item 3 can be deduced from this previous point and the exponential upper bound for the size cost of the simulation of deterministic 1-LA by 1DFA given in [11].  $\square$

### *Acknowledgments*

We are very indebted to Giovanni Pighizzini for suggesting the problem and for many stimulating conversations.

### **References**

- [1] B. Guillon, L. Prigioniero, Linear-time limited automata, in: DCFS 2018, Vol. 10952 of Lecture Notes in Computer Science, Springer, 2018, pp. 126–138.
- [2] J. E. Hopcroft, J. D. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 1979.
- [3] F. C. Hennie, One-tape, off-line Turing machine computations, Information and Control 8 (6) (1965) 553–578.
- [4] K. Tadaki, T. Yamakami, J. C. H. Lin, Theory of one-tape linear-time Turing machines, Theor. Comput. Sci. 411 (1) (2010) 22–43.
- [5] G. Pighizzini, Nondeterministic one-tape off-line Turing machines, Journal of Automata, Languages and Combinatorics 14 (1) (2009) 107–124.
- [6] D. Průša, Weight-reducing Hennie machines and their descriptive complexity, in: LATA 2014, Vol. 8370 of Lecture Notes in Computer Science, 2014, pp. 553–564.

- [7] T. N. Hibbard, A generalization of context-free determinism, *Information and Control* 11 (1/2) (1967) 196–238.
- [8] G. Pighizzini, A. Pisoni, Limited automata and context-free languages, *Fundamenta Informaticae* 136 (1-2) (2015) 157–176.
- [9] M. Kutrib, G. Pighizzini, M. Wendlandt, Descriptive complexity of limited automata, *Inf. Comput.* 259 (2) (2018) 259–276.
- [10] K. W. Wagner, G. Wechsung, *Computational Complexity*, D. Reidel Publishing Company, Dordrecht, 1986.
- [11] G. Pighizzini, A. Pisoni, Limited automata and regular languages, *International Journal of Foundations of Computer Science* 25 (07) (2014) 897–916.
- [12] G. Pighizzini, L. Prigioniero, Limited automata and unary languages, *Information and Computation* doi:<https://doi.org/10.1016/j.ic.2019.01.002>.
- [13] M. Bojańczyk, L. Daviaud, B. Guillon, V. Penelle, Which classes of origin graphs are generated by transducers, in: *ICALP 2017*, Vol. 80 of *LIPIcs*, 2017, pp. 114:1–13.
- [14] J. C. Shepherdson, The reduction of two-way automata to one-way automata, *IBM J. Res. Dev.* 3 (2) (1959) 198–200.
- [15] M. Sipser, Halting space-bounded computations, *Theoretical Computer Science* 10 (3) (1980) 335 – 338.
- [16] V. Geffert, C. Mereghetti, G. Pighizzini, Complementing two-way finite automata, *Inf. Comput.* 205 (8) (2007) 1173–1187.
- [17] C. A. Kapoutsis, Predicate characterizations in the polynomial-size hierarchy, in: *Language, Life, Limits - 10th Conference on Computability in Europe, CiE 2014.*, Vol. 8493 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 234–244.