Nondeterministic forgetting automata are less powerful than deterministic linear bounded automata

Petr Jančar
Dept. of Computer Science, University of Ostrava
Dvořákova 7, 701 03 Ostrava, Czech Republic

A complete proof of a result briefly mentioned in [4] is given.

Forgetting automata are nondeterministic linear bounded automata with restricted rewriting capability: any input symbol can only be "erased" (rewritten by a special symbol) or completely "deleted". They are, in fact, a special case of 2-change automata introduced in [1].

This paper shows by the method of diagonalization that, for any k, k-change automata with a fixed (work) alphabet recognize a proper subclass of the class of languages recognizable by deterministic linear bounded automata (i.e. deterministic context-sensitive languages).

0. Introduction

Forgetting automata were studied e.g. in [3],[4].

The motivation mainly comes from linguistics. It is illustrated by the following simple example (from [4]):

Example. Parsing a sentence can consist in the stepwise leaving out the words whose absence does not affect correctness of the sentence, e.g.

"The little boy ran quickly away"
"The ...... boy ran quickly away"
"The ...... boy ran .... away"

It leads to considering linear bounded automata (LBAs) which are able to rewrite every input symbol by a special symbol only; the operation is referred to as *erasing*.

Such erasing automata were also considered by von Braunmühl and Verbeek in [1] as a special case of finite-change automata. Their motivation was different: to introduce a storage medium "between time and space".

In the example, the segments of erased symbols are not important and can be deleted completely ("The boy ran quickly away", "The boy ran away", ...). It can be modelled by the operation called *deleting*, which "cuts out" the cell (and pastes the two remaining parts of the tape together).

By (the most general) *forgetting automata* we mean nondeterministic LBAs which can only erase and delete cells (but not rewrite them).

Remark. To make deleting more natural, the model of list automata is usually used; a list automaton uses a (doubly linked) list of items (cells) rather than the usual tape.

In [4] all combinations of operations "move the head left (right)", "erase and move left (right)", "delete and move left (right)" were considered and the corresponding classes of languages were classified.

One of the interesting results whose proofs are only sketched in [4] is that nondeterminism with rewriting limited to erasing and deleting is less powerful than (general) determinism: forgetting automata recognize a proper subclass of the class of languages recognizable by deterministic linear bounded automata.

The complete proof of this fact is the main subject of this paper. In fact, the result applies for more general k-change automata (k-CAs) of [1] if we suppose a fixed (work) alphabet. A k-CA can be simulated by a deterministic LBA (DLBA); it is shown in [1] using the well-known idea of Savitch ([5]). We use here the mentioned simulation and complete the proof by the method of diagonalization.

Section 1 contains definitions, Section 2 the result.

1. Definitions

We consider (special cases of) standard *nondeterministic* Turing machines working on a tape consisting of cells. In what follows, we give the needed notation and definitions.

Definition 1 *1.1. Definition A Turing machine, TM, T is a tuple $T = (Q, \Sigma, \Delta, \delta, q_0, F)$, where Q is a finite set of states , Σ is a finite set, the input alphabet, of input symbols, Δ is a finite set, the work alphabet, of work symbols disjoint with Σ, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final (accepting) states and δ is a (finite) set of instructions of the type $[q, a] \longrightarrow [q', a', D]$, where $q, q' \in Q$, $a, a' \in \Sigma \cup \Delta$ (if $a' \neq a$ then $a' \in \Delta$), $D \in \{LEFT, RIGHT\}$ (meaning that T in state q and reading a can change the state to q', rewrite a by a' and move the head in direction D).*

A computation, an accepting computation (starting in the initial configuration and finishing in a final configuration) and the language recognized by T,

$\mathcal{L}(T) \subseteq \Sigma^*$, *as well as the notion of a deterministic TM can be defined in the usual way.*

T is a nondeterministic (deterministic) linear bounded automaton, NLBA (DLBA), if it is a nondeterministic (deterministic) TM and does not move the head outside its input (any input word is bounded by special endmarkers which are never crossed).

The following notion comes from [1].

Definition 2 *1.2. Definition A k-change automaton, k-CA, is a NLBA which rewrites (changes) any cell of its input k-times at most.*

As we mentioned, forgetting automata in [4] and elsewhere were considered as list automata. For our aims it is only important that they can be viewed as 2-change automata (with a fixed work alphabet). They can be (somewhat artificially) defined as follows.

Definition 3 *1.3. Definition A forgetting automaton T is a 2-CA, where the work alphabet consists of two special symbols, say @,#. In addition, # can not be rewritten and can not influence any computation of T (it behaves like deleted, i.e. nonexistent).*

2. Results

The next two lemmas follow from [1]. Nevertheless we show the proofs (perhaps more lucidly) since we need to refer to the way of simulation in the main proof.

Theorem 4 *2.1. Lemma Any 1-CA T_1 can be simulated by a DLBA T_2 (hence $\mathcal{L}(T_1) = \mathcal{L}(T_2)$).*

Proof. A 1-CA is a special case of a NLBA. Hence Savitch's idea ([5]) of simulating a NLBA T_1 by a deterministic quadratic-space bounded TM T_2 applies.

We first outline a (nonrecursive) version of the relevant algorithm. Then we show how the space in our special case (where T_1 is a 1-CA) can be reduced to make T_2 a DLBA.

A NLBA T_1 accepts a given word of length n if it accepts it by a computation of length shorter than 2^{cn}, c being a constant depending on the number q of states of T_1 and the number of work symbols of T_1.

n, q, c keep the mentioned meaning in the rest of the proof. In our context $log(x)$ will mean the least integer greater or equal to the logarithm of x with basis 2.

To find out whether T_1 can reach a final configuration from the initial configuration in less than 2^{cn} steps, T_2 will use space for $cn+1$ configurations (see Fig.1).

It uses consecutive *segments* $S_0, S_1, S_2, \ldots, S_{cn}$ of the tape. Each segment consists of the *state part* of length $log(q)$ for storing the current state, the *head part* of length $log(n)$ for the position of the head and the *tape part* of length n for the content of the tape. $Conf(S_i)$ will denote the configuration stored in S_i.

For technical convenience, segment S_{i+1} is drawn below segment S_i in Fig.1.

Suppose a lexicographic order on the set of all configurations (with the tape of length n). To this order we refer in commands like "$S_i :=$ the least configuration" or "increase S_i" (replace $Conf(S_i)$ by the next greater configuration – if $Conf(S_i)$ is not the greatest). It is clear that T_2 is able to perform such commands.

For a given input word of length n, T_2 writes the initial configuration in S_0 and performs Algorithm 1. Here the last segment (with some property) is the segment S_i with the greatest index (and with the property).

S_0	the initial configuration			$*$
S_1	$log(q)$	$log(n)$	n	$*$
S_2	$log(q)$	$log(n)$	n	
S_3	$log(q)$	$log(n)$	n	$*$
	.....			
S_{cn}	$log(q)$	$log(n)$	n	

Fig.1

S_0	the initial configuration			$*$
S_1	$log(q)$	$log(n)$	n	$*$
S_2	$log(q)$	$log(n)$	$n/2$	
S_3	$log(q)$	$log(n)$	$n/4$	$*$
	.....			
S_k	$log(q)$	$log(n)$	1	$(k = log(n)+1)$

Fig.2

Algorithm 1

Mark S_0 and CLEAR BELOW S_0 ;
while true **do**
  if $Conf(S_i)$, where S_i is the last unmarked segment, can be reached by one step (of T_1) from $Conf(S_j)$, where S_j is the last marked segment
  then mark S_i and CLEAR BELOW S_i ;
    if $Conf(S_i)$ is a final configuration
    then HALT {meaning YES-answer of the algorithm}

else INCREASE
endwhile

where CLEAR BELOW S_i means
for $j := i+1, i+2, \ldots, cn$ **do** $S_j :=$ the least configuration and unmark S_j

and INCREASE means
Find the last S_i which can be increased ;
if $i > 0$ **then** increase S_i, unmark S_i and CLEAR BELOW S_i
else HALT {meaning NO-answer of the algorithm}

Correctness of Algorithm 1 should be clear from the following remarks.
– Before any pass through the cycle, some segments are marked (denoted by $*$ in Fig.1), the others unmarked.
– $Conf(S_1), Conf(S_2), \ldots, Conf(S_{cn})$ are guesses of some configurations "passed through" in a computation. We say that the *guess* for S_i ($i > 0$) is *correct* if $Conf(S_i)$ is reachable by 2^{cn-i} steps from $Conf(S_j)$, where S_j is the previous marked segment (j is the greatest such that $j < i$ and S_j is marked).
– An invariant of the cycle is that guesses in marked segments are always correct.
– The algorithm systematically generates all possible combinations of guesses and verifies their correctness. It halts by encountering a correct (verified) final configuration or by exhausting all possibilities.

Our aim now is to reduce the used space. In a computation of a 1-CA, the content of the tape can be changed n times at most. We define a *multistep* as a sequence of steps where the last step is a "rewriting" one or reaches a final configuration while the others are "nonrewriting" ones. Hence any (finite) computation consists of $n+1$ multisteps at most.

Notice that linear space is surely sufficient to verify whether one configuration can be reached from another by a multistep: it suffices to consider multisteps consisting of qn steps at most. A configuration "passed through" by a given multistep is fully determined by the current state and the position of the head; thus its description can be stored in space $log(q) + log(n)$. Hence the idea of Algorithm 1 can be applied using $log(qn)$ auxiliary segments, each of length $log(q) + log(n)$.

Now it is clear that T_2 simulating a 1-CA T_1 can perform a modification of Algorithm 1 in which steps are replaced by multisteps. Hence approx. $log(n)$ segments are sufficient.

For technical reasons, suppose that n is a power of 2 and that $log(n) + 2$ segments are sufficient (any input word can be appropriately extended by "dummy" symbols, by which its length increases twice at most).

To make the whole space linear, we still have to compress the representation of configurations.

The point is in the following idea.
Suppose (the tape part of) a configuration C; let us refer to the cells with input

(work) symbols as *input (work) cells*. To store a configuration C' which appears (a number of steps) after C in a computation, it suffices to store the cells of C' which correspond to the input cells of C because the contents of the work cells must be the same in C' and C. On the other hand, to store C' which appears (a number of steps) before C, it suffices to store the cells of C' which correspond to the work cells of C.

If $Conf(S_1), Conf(S_2), \ldots, Conf(S_i)$ are correct guesses then the tape part of $Conf(S_i)$ is determined by the tape parts of $Conf(S_1)$, $Conf(S_2)$, …, $Conf(S_{i-1})$ with exception of $n/2^{i-1}$ cells. In these cells in $Conf(S_{i-1})$ are either input symbols – if S_{i-1} is marked – or work symbols – if S_{i-1} is unmarked. (The symbols in) one half out of these $n/2^{i-1}$ cells in $Conf(S_i)$ are the same as in $Conf(S_{i-1})$, the other half of these cells are different – there are work (input) symbols instead of input (work) symbols.

Hence, to store the tape part of S_i, a segment of length $n/2^{i-1}$ is sufficient. An *admissible content* of this segment is any string of $n/2^{i-1}$ symbols out of which one half (i.e. $n/2^i$) are input symbols and the other half are work symbols. Notice also the natural condition of *consistence* – the corresponding input (work) cells in different segments must have the same contents.

These ideas are the basis for the (algorithm of) T_2 for which the segments of lengths shown in Fig.2. (S_i of length $n/2^{i-1}$, for $i > 0$) are sufficient.

T_2 can perform Algorithm 1 (modified for multisteps) provided that it is always able to reconstruct the (complete) tape part of $Conf(S_i)$ for any $i > 1$. It can generate the admissible contents of segments (in the last segment, any symbol is admissible) and use the following (sub)algorithm 2 for the reconstruction as well as for checking the consistence.

Algorithm 2

$A :=$ the tape part of S_i ;
for $j := i-1, i-2, \ldots, 0$ **do**
  $B :=$ the tape part of S_j;
  if S_j is marked
  then replace the input cells in B successively with respective cells in A checking the consistence at the same time {if the symbol in the m-th cell of A is an input one then check that it is the same as the symbol in the m-th input cell of B, otherwise (it is a work symbol) write it in the m-th input cell of B ($m = 1, 2, \ldots, n/2^j$)}
  else replace the work cells in B successively with respective cells in A checking the consistence at the same time
  endif ;
  $A := B$
endfor
{ in the end, A contains the (complete) tape part of $Conf(S_i)$ }

It should be clear that the modified algorithm keeps the correctness (in simulating a 1-CA) and linear space is sufficient for it. Thus the whole proof is finished. □

Theorem 5 *Any k-CA T_1 ($k \geq 1$) can be simulated by a DLBA T_2 (hence $\mathcal{L}(T_1) = \mathcal{L}(T_2)$).*

Proof. A k-CA T_1 starting with an input $a_1 a_2 \dots a_n$ can be simulated by a 1-CA T_1' with input $a_1 ** \dots * a_2 ** \dots * \dots \dots a_n ** \dots *$, where $** \dots *$ means k blank symbols. If T_1 changes the i-th cell for the j-th time then T_1' changes the j-th cell in the i-th block. Then T_2 simulating T_1' (and hence T_1) can be constructed as in the proof of the previous lemma. □

The lemma implies that any language recognized by a k-change automaton (hence also by a forgetting automaton) is recognized by a DLBA. For the next theorem we need to prove that, given any k and any work alphabet Δ, there is a language recognized by a DLBA which is not recognized by any k-CA with the work alphabet Δ. We use the standard method of diagonalization. Let us recall it first (see e.g. [2]).

Consider any encoding of Turing machines by words in alphabet $\{0, 1\}$. If, for some class C of Turing machines, we have a universal (deterministic) TM which halts for any input (deciding if the given encoded TM accepts the given word or not) then there is a "diagonalizing" TM T with input alphabet $\{0, 1\}$ which accepts a code of a TM M from C if and only if M does not accept it. It is clear that $\mathcal{L}(T)$ does not belong to the class $\{L \mid L = \mathcal{L}(M) \text{ for some } M \in C\}$.

For an arbitrary k, consider the class of k-CAs with the input alphabet $\{0, 1\}$ and with a fixed work alphabet. We can choose any natural encoding for this class where the number of states of a k-CA M is bounded by the length of the code of M.

It should be clear from the proofs of previous lemmas how a universal DLBA as well as a diagonalizing DLBA can be constructed ($log(q)$ being replaced by $log(n)$).

Hence we have

Theorem 6 *2.3. Theorem For any k and Δ, the class of languages recognizable by (nondeterministic) k-change automata with the work alphabet Δ is a proper subclass of the class of languages recognizable by DLBAs (i.e. of the class of deterministic context-sensitive languages).*

As any forgetting automaton is a 2-change automaton with the work alphabet $\{@, \#\}$, the theorem implies

Theorem 7 *2.4. Corollary The class of languages recognizable by (nondeterministic) forgetting automata is a proper subclass of the class of languages recognizable by DLBAs.*

Acknowledgements

I would like to thank František Mráz and Martin Plátek who initiated me into work on forgetting automata and encouraged me to write this paper.

References

[1] von Braunmühl B., Verbeek R. Finite change automata Proc. of 4th GI Conference on Theoretical Computer Science, Lecture Notes in Computer Science (LNCS) 67, Springer, Berlin 1979 91–100

[2] Hopcroft J., Ullman J.Formal languages and their relation to automataAddison–Wesley1969

[3] Jančar P., Mráz F., Plátek M. Characterization of context-free languages by erasing automata Proc. of the symp. Mathematical Foundations of Computer Science (MFCS) 1992, Prague, Czechoslovakia, LNCS 629, Springer1992 307–314

[4] Jančar P., Mráz F., Plátek M. A taxonomy of forgetting automata accepted to MFCS'93, Gdańsk, Poland, to appear in LNCS, Springer1993

[5] Savitch W.J. Relationships between nondeterministic and deterministic tape complexities J. of Computer and System Sciences41970177–192