Computing the prefix of an automaton

MARIE-PIERRE BÉAL
Institut Gaspard Monge*
Université de Marne-la-Vallée

OLIVIER CARTON
Institut Gaspard Monge*
Université de Marne-la-Vallée

March 7, 2001

Abstract

We present an algorithm for computing the prefix of an automaton. Automata considered are non-deterministic, labelled on words, and can have ε-transitions. The prefix automaton of an automaton $\mathcal{A}$ has the following characteristic properties. It has the same graph as $\mathcal{A}$. Each accepting path has the same label as in $\mathcal{A}$. For each state q, the longest common prefix of the labels of all paths going from q to an initial or final state is empty. The interest of the computation of the prefix of an automaton is that it is the first step of the minimization of sequential transducers.

The algorithm that we describe has the same worst case time complexity as another algorithm due to Mohri but our algorithm allows automata that have empty labelled cycles. If we denote by $P(q)$ the longest common prefix of labels of paths going from q to an initial or final state, it operates in time $O((P+1) \times |E|)$ where P is the maximal length of all $P(q)$.

1 Introduction

Transducers are finite state machines whose transitions or edges are labelled by a pair made of an input word and an output word. They are widely used in practice to model various things like lexical analyzers in language processing [14], operations in numeration systems [11] or also encoding or decoding schemes for channels [2]. As a transducer has input and output labels, and even if these labels are letters, there is in general no minimal equivalent object like for simple finite state automata. It is very often required that the transducer has letters as input labels and has moreover a deterministic input automaton. It is then called sequential. Used as an encoder, this means that the output codeword is obtained sequentially from the input data. Transducers which are not sequential, but which realize sequential functions, can be first determinized (see for instance [4]

*Institut Gaspard Monge, Université de Marne-la-Vallée, 5 boulevard Descartes, 77454 Marne-la-Vallée Cedex 2, France. http://www-igm.univ-mlv.fr/~{beal,carton}

or [3]). In the case of sequential transducers, there exists a minimal equivalent sequential transducer, even if the output labels are variable length words.

A characterization of minimal sequential transducers was first given in [7]. A procedure to produce a minimal sequential transducer is there indicated. It is in particular shown in [7] that the minimal sequential transducer is obtained in two steps. The first one is the computation of the prefix automaton of the output automaton of the transducer. The second step is a classical minimization of the transducer obtained at the end of the first step, seen as an ordinary finite state automaton. The prefix of an automaton can be interpreted as an automaton with the same underlying graph, same behaviour but produces its output as soon as possible. Its name comes from the fact that for any state q, the longest common prefix $P(q)$ of labels of paths going from q to an initial or final state is empty.

The first algorithm of computation of the prefix of an automaton appears in [12] and [13]. The construction is there called a quasi-determinization. It has been noticed by Mohri that the first step of the minimization of sequential transducers is independent from the notion of transducers. The quasi-determinization is an algorithm that works on finite state automata. It keeps the graph of an automaton and changes only the labels of the edges. Roughly speaking, it pushes the labels of the edges from the final states towards the initial states as much as possible. The algorithm of Mohri has a time complexity $O((P+1) \times |E|)$, where E is the set of edges and P the maximum of the lengths of $P(q)$ for all states q. We assume here that the number of states $|Q|$ is less than the number of edges. Another algorithm for computing the prefix of automaton has been presented in [5] and [6]. The approach of this algorithm is really different from ours. It is based on the construction of the suffix tree of a tree and its time complexity is $O(|Q|+|E|+S \log |A|)$, where A is the alphabet and S is the sum of the lengths of the labels of all edges of the automaton. Breslauer's algorithm can thus be better when there is a small number of edges and Mohri's algorithm is better in the other case. In practice, S can be very large and P can be very small. This makes the algorithms of Mohri and ours almost linear. A comparison of the two complexities is given in [13].

Our algorith uses the same principle of pushing letters through states as Mohri's algorithm does. Main restriction to Mohri's algorithm is that it does not work when the automaton contains a cycle of empty label (the system of equations given in [13, Lemma 2 p. 182] does not admits a unique solution in this case). Some step in Mohri's algorithm requires that the automaton has no empty labelled cycle. However, if the starting automaton does not have any such cycle, this property is kept along the process. The algorithm is therefore correct in this case. This restriction is not really important for applications since the transducers used in practice, like in language processing, have no empty labelled cycles in output.

In this paper, we present another algorithm of computation of the prefix of an automaton which has the same worst case time complexity as Mohri's algorithm, $O((P+1) \times |E|)$, and that works for all automata. The existence of empty labelled cycles accounts for most of the difficulty in the coming algorithm. The time complexity is independent of the size of the alphabet. The algorithm consists in decreasing by 1 the value P at each step. We present our algorithm for sequential transducers but it can be directly extended to the case of subsequential transducers (see [7] or [4] for the definition of a subsequential transducer).

In Section 2, we recall some basic definitions from automata theory and we define the prefix automaton of an automaton. The computation algorithm of the prefix of an automaton is presented in Section 3. The complexity is analyzed in Section 4. In that section some data structures are described which can be used to get the right time complexity of the algorithm.

2 Prefix of an automaton and applications

In the sequel, A denotes a finite alphabet and ε is the empty word. A word u is a *prefix* of a word v if there is a word w such that $v = uw$. The word w is denoted by $u^{-1}v$. The *longest common prefix* of a set of words is the longest word which is prefix of all words of the set.

An *automaton* over A^* is composed of a set Q of *states*, a set $E \subset Q \times A^* \times Q$ of *edges* and two sets $I, F \subset Q$ of *initial* and *final* states. An edge $e = (p, u, q)$ from p to q is denoted by $p \xrightarrow{u} q$, the word u being the label of the edge. The automaton is finite if Q and E are finite. A *path* is a possibly empty sequence of consecutive edges. Its label is the concatenation of the labels of the consecutive edges. An automaton is often denoted by $\mathcal{A} = (Q, E, I, F)$. An *accepting path* is a path from an initial state to a final state. The language or set of words recognized (or accepted) by an automaton is the set of labels of accepting paths. An automaton is *deterministic* if it is labelled by letters of a finite alphabet A, if it has one initial state and if for each state p and each letter a in A, there is at most one edge $p \xrightarrow{a} q$ for some q.

We now define the prefix automaton of a given automaton $\mathcal{A}$. This prefix automaton has the same graph as $\mathcal{A}$, but the labels of the edges are changed. However the labels of the accepting paths remain unchanged and the prefix automaton recognizes the same words. Furthermore, for any state q of the prefix automaton the longest common prefix of the labels of all paths going from q to an initial or final state is empty.

Let $\mathcal{A} = (Q, E, I, F)$ be a finite non-deterministic automaton labelled by words. We assume that the automaton is *trim*, that is, any state belongs to an accepting path. For each state q, we denote by $P_{\mathcal{A}}(q)$, or just $P(q)$, the longest common prefix of the labels of all paths going from q to an initial or final state. Remark that $P(q) = \varepsilon$ if q is initial or final.

The *prefix automaton* of $\mathcal{A}$ is the automaton $\mathcal{A}' = (Q, E', I, F)$ defined as follows.

$$E' = \{q \xrightarrow{P(q)^{-1}uP(r)} r \mid q \xrightarrow{u} r \text{ is an edge of } \mathcal{A}\}.$$

One may easily check that if $q \xrightarrow{u} r$ is an edge of $\mathcal{A}$, then the word $P(q)$ is by definition a prefix of the word $uP(r)$ and the previous definition is thus consistent.

Note that a path labelled by w from q to r in $\mathcal{A}$ becomes a path labelled by $P(q)^{-1}wP(r)$ from q to r in the prefix automaton. If this path is accepting, q is initial and r is final and thus $P(q)$ and $P(r)$ are both empty. Then the label of the path in the prefix automaton is the same as in $\mathcal{A}$. The label of a cycle of $\mathcal{A}$ is conjugated to its label in the prefix automaton. In particular the empty labelled cycles of the prefix automaton are the same as the ones of $\mathcal{A}$.

By construction the longest common prefix of the labels of all paths going from q to an initial or final state is empty in the prefix automaton.

Our definition of the prefix automaton allows edges coming in an initial state. In most cases, there is none and for each non-initial state q, $P(q)$ is the longest common prefix of the labels of all paths going from q to a final state.

The words $P(q)$ are the longest words such that $P(q) = \varepsilon$ if q is initial or final and such that $P(q)$ is a prefix of $uP(r)$ for any edge $q \xrightarrow{u} r$. Indeed, if a function P' maps any state q to a word such that these two conditions are met, then $P'(q)$ is a prefix of $P(q)$ for any state q.

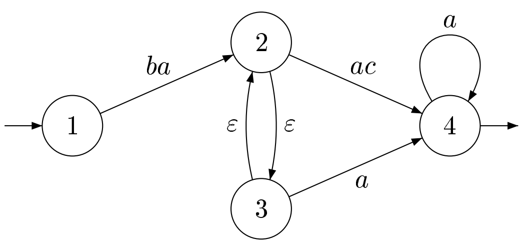

Figure 1: An automaton $\mathcal{A}$.

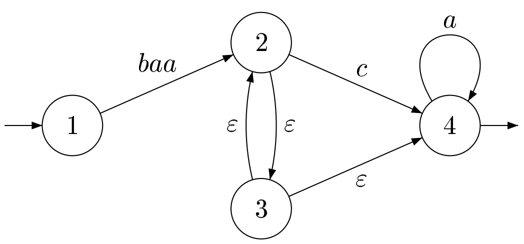

Figure 2: The prefix automaton of $\mathcal{A}$.

EXAMPLE 1 Consider the automaton $\mathcal{A}$ pictured in Figure 1 where the initial state is 1 and the final state is 4. The prefix automaton of $\mathcal{A}$ is pictured in Figure 2.

The main application of the prefix of an automaton is minimization of sequential and subsequential transducers. A transducer is defined as an automaton, except that the labels of the edges are pairs made of an input word and an output word. A transducer labelled in $A \times B^*$ is *sequential* if its input automaton is deterministic. It has been proved [7], [8, p. 95], see also [12] and [13], that among the sequential transducers computing a given function, there is a minimal one which can be obtained from any sequential transducer computing the function. This minimization is performed in two steps. The first step is the computation of the prefix automaton of the output automaton of the transducer. The second step is a minimization of the resulting transducer, considered as a finite automaton.

We refer to [12] for examples of minimization of sequential transducers.

3 Computation of the prefix of an automaton

In this section, we describe an algorithm which computes the prefix of an automaton. The automaton $\mathcal{A} = (Q, E, I, T)$ is a non-deterministic automaton whose edges are labelled by words over a finite alphabet A. The labels can be the empty word and cycles with empty labels are allowed.

We first describe the principle of the algorithm. If q is a state of $\mathcal{A}$, we recall that $P(q)$ denotes the longest common prefix of the labels of all paths going from q to an initial or final state. We denote by $p(q)$ the first letter of $P(q)$ if $P(q) \neq \varepsilon$, and ε if $P(q) = \varepsilon$.

We denote by $P_{\mathcal{A}}$ the maximum of the lengths of all $P(q)$ for all states q.

If $P_{\mathcal{A}} > 0$, we construct from the automaton $\mathcal{A} = (Q, E, I, T)$ an automaton $\mathcal{A}' = (Q, E', I, T)$ whose edges are defined as follows:

$$E' = \{q \xrightarrow{p(q)^{-1}up(r)} r \mid q \xrightarrow{u} r \text{ is an edge of } \mathcal{A}\}.$$

It recognizes the same language as $\mathcal{A}$ and satisfies $P_{\mathcal{A}'} = P_{\mathcal{A}} - 1$. By iterating this process, we get the prefix automaton.

We now explain the computation of the automaton $\mathcal{A}'$. We call ε-edge any edge whose label is ε. Let $\mathcal{A}_\varepsilon$ be the sub-automaton of $\mathcal{A}$ obtained by keeping only the ε-edges. We first compute the strongly connected components of $\mathcal{A}_\varepsilon$. This can be performed by depth-first explorations of $\mathcal{A}_\varepsilon$ [9]. The strongly connected components are stored in an array c indexed by Q. For each state q we denote by $c[q]$ a state that represents the strongly connected component of q. The call to STRONGLY-CONNECTED-COMPONENTS($\mathcal{A}_\varepsilon$) in the pseudo code below will refer to this procedure that computes the array c.

Note that all states q in a same strongly connected component of $\mathcal{A}_\varepsilon$ have same $P(q)$ and thus same $p(q)$.

The construction of $\mathcal{A}'$ is then done with two depth-first explorations, first an exploration of $\mathcal{A}_\varepsilon$, second, an exploration of $\mathcal{A}$.

The first exploration computes $p(q)$ for each state q of $\mathcal{A}_\varepsilon$. This symbol, either a letter or ε, is stored in the cell *letter*$[q]$ of an array *letter*. As $p(q)$

is common to all states q in a same strongly connected component of $\mathcal{A}_\varepsilon$, we compute it only for the states $c[q]$.

At the beginning of the computation, all cells *letter*$[q]$ are set to the default value $\top$ which stands for undefined. During the computation, these values are changed into symbols of $A \cup \{\varepsilon\}$. Let X be the set $A \cup \{\varepsilon, \top\}$. We define a partial order on the set X as follows. For each $a \in A$,

$$\varepsilon < a < \top.$$

Note that each subset of X has an inf in X such that, for all $x \in X$, all $a, b \in A$ with $a \neq b$,

$$\begin{aligned}\inf(\varepsilon, x) &= \varepsilon,\\ \inf(\top, x) &= x,\\ \inf(a, b) &= \varepsilon.\end{aligned}$$

We also assume that an array *local* indexed by Q gives, for each state q, either ε if q is final or initial, or $\inf(S)$ where S is the set of letters that appear as the first letter of a non-empty label of an edge going out of q. Note that if there is no edge with a non-empty label going out of q, *local*$[q]$ is equal to $\top$. The array *local* is initialized by the procedure INIT-TABLE and updated with the procedures UPDATE-TABLE-HEAD and UPDATE-TABLE-TAIL that we shall describe later.

For each state q in Q, the value of *letter*$[c[q]]$ is first set to the inf of *local*$[r]$, for all states r in the same strongly connected component of $\mathcal{A}_\varepsilon$ as q. This is done by the procedure INIT-LETTER. During the exploration of the automaton $\mathcal{A}_\varepsilon$, if q has a successor r such that *letter*$[c[r]] <$ *letter*$[c[q]]$, then *letter*$[c[q]]$ is changed in $\inf($*local*$[q]$, *letter*$[c[q]])$. We claim that the cell of index q of the array *letter* contains $p(q)$ at the end of this exploration. This exploration is done by the function FIND-LETTER. It returns a boolean which is true if there is at least one state q with $p(q)$ non-empty.

We give below a pseudo code for the procedures INIT-LETTER, FIND-LETTER and FIND-LETTER-VISIT. We follow the depth-first search presentation of [9].

INIT-LETTER(set of states Q)
    for each state $q \in Q$ **do**
        letter$[c[q]] \leftarrow \top$
    for each state $q \in Q$ **do**
        letter$[c[q]] \leftarrow \inf($*local*$[q]$, *letter*$[c[q]])$

FIND-LETTER(automaton $\mathcal{A}_\varepsilon = (Q, E_\varepsilon, I, F)$)
    bool $\leftarrow$ FALSE
    for each state $q \in Q$ **do**
        color$[q] \leftarrow$ WHITE
    for each state $q \in Q$ **do**
        if *color*$[q] =$ WHITE **then**
            FIND-LETTER-VISIT$(\mathcal{A}_\varepsilon, q)$
    return *bool*

FIND-LETTER-VISIT(automaton $\mathcal{A}_\varepsilon = (Q, E_\varepsilon, I, F)$, state q)
    $color[q] \leftarrow$ BLACK
    for each edge (q, ε, r) **do**
        if $color[r] =$ WHITE **then**
            FIND-LETTER-VISIT($\mathcal{A}_\varepsilon, r$)
        $letter[c[q]] \leftarrow \inf(letter[c[q]], letter[c[r]])$
    if $letter[c[q]] \neq \varepsilon$ **then**
        $bool \leftarrow$ TRUE

We now prove the correctness of our algorithm.

PROPOSITION 2 *Function* FIND-LETTER *computes $p(q)$ for each state q.*

Proof. For each state q, "$letter[c[q]] \geq p(q)$" is an invariant of the function FIND-LETTER. Indeed, one has $local[r] \geq p(q)$, for each state r in the same strongly connected component as q. This implies that "$letter[c[q]] \geq p(q)$" is an invariant of the function INIT-LETTER(Q). Moreover, if there is an edge (q, ε, r) and if $letter[c[r]] \geq p(r)$, we get $letter[c[r]] \geq p(r) \geq p(q)$. Then "$letter[c[q]] \geq p(q)$" is invariant during FIND-LETTER-VISIT($\mathcal{A}_\varepsilon, q$).

We now show that if there is an edge (q, ε, r) between two states q and r, we have $letter[c[q]] \leq letter[c[r]]$ at the end of FIND-LETTER($\mathcal{A}_\varepsilon$). This fact is trivial if q and r belong to the same strongly connected component of $\mathcal{A}_\varepsilon$. If not, the end of the exploration of state r is before the end of the exploration of q. Then the line 5 of FIND-LETTER-VISIT($\mathcal{A}_\varepsilon, q$) implies that $letter[c[q]] \leq letter[c[r]]$.

Let us assume there is a (possibly empty) path from q to a state r which has an empty label and an edge going out of r labelled with au, where u is a word. Then $letter[c[q]] \leq a$ at the end of FIND-LETTER-VISIT($\mathcal{A}_\varepsilon, q$). Indeed, at the end of FIND-LETTER-VISIT($\mathcal{A}_\varepsilon, q$), we have $letter[c[r]] \leq a$, and then also $letter[c[q]] \leq letter[c[r]] \leq a$.

Let us assume that $p(q)$ is a letter a in A. Then there is a (possibly empty) path from q to a state r which has an empty label and an edge going out of r labelled with au, where u is a word. As a consequence $letter[c[q]] \leq a$ and then $letter[c[q]] = p(q)$. Let us now assume that $p(q)$ is the empty word. Then there is either a (possibly empty) path from q to a state r which has an empty label and an edge going out of r labelled with au, where u is a word, and there is a (possibly empty) path from q to a state r' which has an empty label and an edge going out of r' labelled with bu, where u is a word, with $b \neq a$. In this case $letter[c[q]] \leq \inf(a, b) = \varepsilon$, and then $letter[c[q]] = p(q)$. Or there is a (possibly empty) path from q to a state r which has an empty label and with r final or initial. Again $letter[c[q]] \leq letter[c[r]] = \varepsilon$. Finally, $letter[c[q]] = p(q)$ for each q. □

The second depth-first exploration is an exploration of the automaton $\mathcal{A}$. It updates the labels of $\mathcal{A}$ in order to decrease the length of $P(q)$ for each state q such that $p(q)$ is non-empty. For each edge (q, u, r), where u is a finite word, the following two operations are performed. The letter (or empty word) $p(c[r])$ is added at the end of u. Then the first letter (or empty word) $p(c[q])$ is removed from the beginning of u. Note that these two operations are possible. If u is nonempty, then $p(c[q])$ is the first letter of u and if $u = \varepsilon$ then $p(c[q]) = p(c[r])$ or $p(c[q]) = \varepsilon$. These operations change the labels of the edges of the automaton $\mathcal{A}$ and thus also the values of the array *local*. Lines 3 and 5 of MOVE-LETTER-VISIT change the labels of the edge e in $\mathcal{A}$. Since an edge with empty label can become an edge with a non-empty label and conversely, the edge of $\mathcal{A}_\varepsilon$ are also updated there. The values of the array *local* are updated with two procedures UPDATE-TABLE-HEAD and UPDATE-TABLE-TAIL described later. The exploration is done during the run of procedure MOVE-LETTER whose pseudo code is given below.

MOVE-LETTER(automaton $\mathcal{A} = (Q, E, I, F)$)
    for each state $q \in Q$ **do**
        color[q] $\leftarrow$ WHITE
    for each state $q \in Q$ **do**
        if *color*[q] $=$ WHITE **then**
            MOVE-LETTER-VISIT($\mathcal{A}, q$)

MOVE-LETTER-VISIT(automaton $\mathcal{A} = (Q, E, I, F)$), state q)
    color[q] $\leftarrow$ BLACK
    for each edge $e = (q, u, r)$ where u is a (possibly empty) word **do**
        append *letter*[$c[r]$] at the end of the label of e in $\mathcal{A}$ and update $\mathcal{A}_\varepsilon$
        UPDATE-TABLE-TAIL(e, *letter*[$c[r]$])
        remove *letter*[$c[q]$] from the head of the label of e in $\mathcal{A}$ and update $\mathcal{A}_\varepsilon$
        UPDATE-TABLE-HEAD(e, *letter*[$c[q]$])
        if *color*[r] $=$ WHITE**then**
            MOVE-LETTER-VISIT($\mathcal{A}, r$)

PROPOSITION 3 *Function transforms the automaton $\mathcal{A}$ in an automaton $\mathcal{A}'$ whose edges are:*

$$E' = \{q \xrightarrow{p(q)^{-1}up(r)} r \mid q \xrightarrow{u} r \text{ is an edge of } \mathcal{A}\}.$$

Therefore, the function MOVE-LETTER *changes the label w of any path from q to r into $p(q)^{-1}wp(r)$.*

Proof. This follows directly from the construction. □

PROPOSITION 4 *Function* MOVE-LETTER *transforms the automaton $\mathcal{A}$ in an automaton $\mathcal{A}'$ which has the same graph as $\mathcal{A}$, keeps the labels of accepting paths and satisfies $P_{\mathcal{A}'} = P_{\mathcal{A}} - 1$.*

Proof. Let w be the label of a path from an initial state i to a final state t in $\mathcal{A}$. The label of the same path obtained at the end of MOVE-LETTER in $\mathcal{A}'$ is $p(i)^{-1}wp(t) = w$. Thus the labels of accepting paths are unchanged. Moreover, for each state q one has $P_{\mathcal{A}'}(q) = p_{\mathcal{A}}(q)^{-1}P_{\mathcal{A}}(q)$. It follows that $P_{\mathcal{A}'} = P_{\mathcal{A}} - 1$ if $P_{\mathcal{A}} \geq 1$. □

We now give a pseudo code of the procedure MAKE-PREFIX which is the main procedure of the algorithm.

MAKE-PREFIX(automaton $\mathcal{A} = (Q, E, I, F)$)
    INIT-TABLE($\mathcal{A}$)
    STRONGLY-CONNECTED-COMPONENTS($\mathcal{A}_\varepsilon$)
    repeat
        INIT-LETTER(Q)
        bool $\leftarrow$ FIND-LETTER($\mathcal{A}_\varepsilon$)
        if *bool* **then**
            MOVE-LETTER($\mathcal{A}$)
    until *bool* = FALSE

The result of the computation of the automaton $\mathcal{A}$ pictured in Figure 1 is the automaton pictured in Figure 2. The automaton $\mathcal{A}$ is such that $P_\mathcal{A} = \varepsilon$. Note that this automaton has an empty labelled cycle.

REMARK 5 The two procedures FIND-LETTER and FIND-LETTER-VISIT can be performed on the directed acyclic graph obtained as the quotient of $\mathcal{A}_\varepsilon$ by the relation of being in a same strongly connected component. This graph can be much smaller than $\mathcal{A}_\varepsilon$ itself. It can be computed by the procedure STRONGLY-CONNECTED-COMPONENTS.

REMARK 6 By proposition 3, the label of a cycle is changed into one of its conjugate by the function MOVE-LETTER. Therefore, the strongly connected components of $\mathcal{A}_\varepsilon$ are unchanged during the iteration of function MAKE-PREFIX.

4 Data structures and complexity

In order to analyze the complexity of our algorithm, we briefly discuss a possible implementation of structures required in the construction.

A classical way for implementing the automaton $\mathcal{A}$ is to use $|Q|$ adjacency lists that represent the edges. We may assume that we have two adjacency lists for each state q. The first one represents the edges of empty label going out of q, that is the edges that also belong to $\mathcal{A}_\varepsilon$. The second one represents the edges of non-empty label going out of q.

In order to compute, for each state q, *local*(q) in a constant time, we maintain an array L indexed by Q defined as follows:

- $L[q]$ is the list of pairs (a, n) with $a \in A, n > 0 \in \mathbb{N}$, such that q has at least one outgoing edge labelled by a word whose first letter is a and such that n is the positive number of edges going out of q and whose first letter is a.

We point out that the first component of an element of $L[q]$ is a letter and never contains ε. Thus *local*(q) is ε if $L[q]$ has more than one element or if q is initial or final. It is the letter a if $L[q]$ contains exactly one pair (a, n) and q is neither initial nor final. It is $\top$ otherwise.

The operation performed in the lists are the insertion of a new letter, that is a pair $(a, 1)$, the incrementation and decrementation of the second component of an element, and the deletion of a letter, that is of a pair $(a, 1)$. We need all these operations to be performed in a constant time.

We use a known technique which allows us to get this time complexity (see for example [1] exercise 2.12 p. 71 and [10] exercise "Implantation de fonctions partielles" 1.14 Chapter 1). This technique is based on the use of array of size $|Q| \times |A|$ which is not initialized.

We assume that the lists $L[q]$ are doubly linked and implemented with cursors. We denote by T an array of variable size. The cells of T are used to store the elements of the lists $L[q]$. Each cell has several fields: a field *label* which contains the letter, a field *number* that contains the number of edges going out of q whose first letter is *label*, a field *state* which contains the state q such that the cell belongs to $L[q]$, and finally fields *next* and *prev* that give the index of the next (respectively previous) element in the same list. The cell of index q of the array L is the index in T of the first element of $L[q]$, if this list is non-empty.

Another array U, indexed by $Q \times A$, gives for each pair (q, a) the index in T of the cell of $L[q]$ whose letter is a, if this letter is in $L[q]$. This array allows us to access an element of a list in a constant time. The operations of insertion, deletion of an element in a list are then done in a constant time. The operations of incrementation and decrementation of the field *number* of the cell of a given label in a given list are also done in a constant time. Indeed, to increment the field *number* of the letter a in $L[q]$, one increments the field *number* of the cell of T indexed by $U[q, a]$.

The array T is initially empty and its size is 0. The size of T is incremented when a new cell is needed in T. A cell that corresponds to an element of a list that has just been removed is marked to be free. Thus the existence of a letter a in $L[q]$ is obtained by checking whether $U[q, a]$ is an index i in $[1, \mathrm{size}(T)]$, whether the cell $T[i]$ is not marked free, and whether the fields *label* and *state* are respectively equal to a and q. This is performed in a constant time.

All the lists of successors that represent the edges of the automaton $\mathcal{A}$ and $\mathcal{A}_\varepsilon$, and the arrays *local*, L, T, U are updated when the label of an edge is changed during the process. The arrays L and *local* are initialized by the procedure Init-Table. The arrays L, T, U and *local* are updated by the procedures Update-Table-Head and Update-Table-Tail.

We give below a pseudo code for the procedure Init-Table.

Init-Table(automaton $\mathcal{A} = (Q, E, I, F)$)
    for each $q \in Q$ **do**
        $L[q] \leftarrow$ the empty list
        local$[q] \leftarrow \top$
    for each $q \in Q$ **do**

for each edge (q, au, r) where a is letter and u a word **do**
    if a is not in $L[q]$ **then**
        insert the pair $(a, 1)$ in $L[q]$
    else increment the field *number* of the letter a in $L[q]$
if $L[q]$ has more than one element or if q is initial or final **then**
    $local[q] \leftarrow \varepsilon$
else if $L[q]$ is not empty **then**
    $local[q] \leftarrow$ the unique letter of $L[q]$

We now describe the updating of the tables and lists. An update is needed as soon as the label of an edge of $\mathcal{A}$ is changed. Note that the labels of the edges of the automata $\mathcal{A}$ and $\mathcal{A}_\varepsilon$ are changed in a constant time. Indeed, a label of an edge going out of a state q that becomes empty is removed from the list of edges of non-empty labels going out of q, and added into the list of edges of empty labels going out of q (and conversely). This is performed in a constant time in line 3 and line 5 of Move-Letter-Visit. To update the arrays L, T, U and *local*, we distinguish the two kinds of modification of the labels of the edges. A letter or the empty word can be added at the end of a label. The procedure called to update is in this case the procedure Update-table-Tail. A letter or the empty word can be removed from the head of the label. The procedure called to update is in this case the procedure Update-table-Head.

Pseudo codes for Update-table-Tail and Update-table-Head are given below.

Update-table-Tail(edge $e = (q, u, r)$, letter (or empty word) x)
    if $u = \varepsilon$ and $x \neq \varepsilon$ **then**
        if x is not in $L[q]$ **then**
            insert the pair $(x, 1)$ in $L[q]$
        else increment the field *number* of the letter x in $L[q]$
        if $L[q]$ has more than one element or if q is initial or final **then**
            $local[q] \leftarrow \varepsilon$
        else $local[q] \leftarrow$ the unique letter of $L[q]$

Update-table-Head(edge $e = (q, u, r)$, letter (or empty word) x)
    We have $u = xu'$, where u' is a finite word, whenever $x \neq \varepsilon$
    if $x \neq \varepsilon$ **then**
        decrement the field *number* of the letter x in $L[q]$
        if this field is equal to 0 **then**
            remove the pair $(x, 0)$ from $L[q]$
        if $u' = bu''$ where b is a letter of A **then**
            if b is not in $L[q]$ **then**
                insert the pair $(b, 1)$ in $L[q]$
            else increment the field *number* of the letter b in $L[q]$
        if $L[q]$ has more than one element or if q is initial or final **then**
            $local[q] \leftarrow \varepsilon$
        else if $L[q]$ has exactly one element **then**
            $local[q] \leftarrow$ the unique letter of $L[q]$

else *local*$[q] \leftarrow \top$

We analyze now the complexity of our algorithm. We denote by $|S|$ the cardinality of a set S. As the automaton is trim, $|Q| \leq |E| + 1$. We also denote by $|E_\varepsilon|$ the cardinality of the current automaton $\mathcal{A}_\varepsilon$. We always have $|E_\varepsilon| \leq |E|$ but the automaton $\mathcal{A}_\varepsilon$ may be much smaller than $\mathcal{A}$. We denote here by P the maximal length of the words $P(q)$ for all states q.

PROPOSITION 7 *Function* MAKE-PREFIX *works in time* $O((P+1) \times |E|)$.

Proof. Function INIT-TABLE can be implemented to work in time $O(|Q| + |E|)$. Functions STRONGLY-CONNECTED-COMPONENTS and FIND-LETTER can be implemented to work in time $O(|Q| + |E_\varepsilon|)$. Function INIT-LETTER works in time $O(|Q|)$. As discussed above, function UPDATE-TABLE works in time $O(1)$.

Function MOVE-LETTER works in time $O(|Q| + |E|)$. Finally the loop in MAKE-PREFIX is executed at most $P+1$ times. The complexity of our algorithm is then $O((|Q| + |E|) \times (P+1) + (|Q| + |E_\varepsilon|) \times (P+1))$. Since the automata considered are trim, $|Q| \leq |E| + 1$ and the complexity is thus $O((P+1) \times |E|)$. □

Let S be the sum of the lengths of the labels of all edges of the automaton. The space complexity of the algorithm is $O((|Q| \times |A|) + |E| + S)$.

5 Acknowledgements

We thank Christian Choffrut and Maxime Crochemore for useful discussions and comments. Christian Choffrut pointed out to us the inaccuracy of the algorithm of [13] in the particular case where the automaton has an empty labelled cycle. We also thank the anonymous referees for their relevant remarks.

References

[1] AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.

[2] BÉAL, M.-P. *Codage Symbolique.* Masson, 1993.

[3] BÉAL, M.-P., AND CARTON, O. Determinization of transducers over finite and infinite words. Tech. Rep. 99-12, I.G.M., Université de Marne-la-Vallée, 1999.

[4] BERSTEL, J. *Transductions and Context-Free Languages.* B.G. Teubner, 1979.

[5] BRESLAUER, D. The suffix tree of a tree and minimizing sequential transducers. In *CPM'96* (1996), vol. 1075 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 116–129.

[6] Breslauer, D. The suffix tree of a tree and minimizing sequential transducers. *Theoret. Comput. Sci.*, 191 (1998), 131–144.

[7] Choffrut, C. *Contribution à l'étude de quelques familles remarquables de fonctions rationnelles.* Thèse d'État, Université Paris VII, 1978.

[8] Choffrut, C. A generalization of Ginsburg and Rose's characterization of gsm mappings. In *ICALP'79* (1979), vol. 71 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 88–103.

[9] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms.* MIT Press, 1990.

[10] Crochemore, M., Hancart, C., and Lecroq, T. *Algorithmique du Texte.* Vuibert, 2000. to appear.

[11] Frougny, C. Numeration systems. In *Algebraic Combinatorics on Words*, M. Lothaire, Ed. Cambridge, 2000. to appear.

[12] Mohri, M. Minimization of sequential transducers. In *CPM'94* (1994), M. Crochemore and D. Gusfield, Eds., vol. 807 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 151–163.

[13] Mohri, M. Minimization algorithms for sequential transducers. *Theoret. Comput. Sci.*, 234 (2000), 177–201.

[14] Roche, E., and Schabes, Y. *Finite-State Language Processing.* MIT Press, Cambridge, 1997, ch. 7.