Parsing with a finite dictionary

Julien Clément [a] , Jean-Pierre Duval [b] , Giovanna Guaiana [b] , Dominique Perrin [a] and Giuseppina Rindone [a]

[a] *Institut Gaspard-Monge, Université de Marne-la-Vallée, France*

[b] *LIFAR, Université de Rouen, France*

Abstract

We address the following issue: given a word $w \in A^*$ and a set of n nonempty words X, how to determine efficiently whether $w \in X^*$ or not? We discuss several methods including an $O(r \times |w| + |X|)$ algorithm for this problem where $r \leq n$ is the length of a longest suffix chain of X and $|X|$ is the sum of the lengths of words in X. We also consider the more general problem of providing all the decompositions of w in words of X.

Key words: Finite automata, string matching

1 Introduction

The complexity of algorithms related to finite automata and regular expressions is well-known in general. In this article, we focus on a particular problem, namely the complexity of parsing a regular language of the form $Y = X^*$ where X is a finite set of nonempty words. This type of language occurs often in the applications, when X is a dictionary and Y is the set of texts obtained by arbitrary concatenations of strings from this dictionary. The time and space complexity can be an important issue in such applications since the dictionaries used for natural languages can contain up to several million words.

As a consequence of general constructions from automata theory, any regular language can be parsed in time proportional to the product of the size of the

Email addresses: Julien.Clement@univ-mlv.fr (Julien Clément), Jean-Pierre.Duval@univ-rouen.fr (Jean-Pierre Duval), Giovanna.Guaiana@univ-rouen.fr (Giovanna Guaiana), Dominique.Perrin@univ-mlv.fr (Dominique Perrin), Giuseppina.Rindone@univ-mlv.fr (Giuseppina Rindone).

regular expression by the length of the input word. This just amounts to simulating a nondeterministic automaton built in a standard way from the regular expression. Using a deterministic automaton produces a linear-time algorithm after completing a determinization algorithm which may itself be exponential.

Our main result here is an algorithm allowing to parse a regular language of the form X^*, with X finite, in time $O(r \times |w| + |X|)$ with r the length of a longest suffix chain in X, w the input word and $|X|$ the sum of the lengths of words in X (Sections 4 and 5). The quantity r depending only on the set X is upper-bounded by $\mathrm{Card}(X)$. This algorithm allows to get all the decompositions of the input word in words of X. We also discuss some further problems on the automata related to regular expressions of this type (Section 3).

The motivation of our study is in the work of Schützenberger on this type of languages. He has shown that although the size of the automaton depends on the length of the words in X, several syntactic parameters depend only on the cardinality of X (see [1]). One of them is linked with the number of interpretations of a word in words of X, and is related with the problem considered here.

A similar yet unsolved problem is the complexity of the problem of unambiguity of the expression X^*, i.e. of the problem of testing whether X is a code. The standard algorithm [2] gives a quadratic complexity $O(|X|^2)$ where $|X|$ is the total length of the words of X. It was lowered later to $O(\mathrm{Card}(X) \times |X|)$ by various authors [3–6]. However it is not known whether there exists a linear algorithm (see [7]).

2 Preliminaries and notations

For a more complete description of automata and fundamentals of formal languages, the reader is referred to [8–10] and to [11] in particular for a recent overview on recognizable languages in free monoids.

Let A be a finite alphabet. We denote by ε the *empty word* and by A^* (resp. A^+) the set of finite words (resp. nonempty finite words) on A. For a word $w \in A^*$, we denote by $|w|$ the length of w, by $w[j]$ for $0 \leq j < |w|$ the letter of index j in w, and by $w[j..k] = w[j]w[j+1]\cdots w[k]$. For any decomposition $w = uv$ with $u, v \in A^*$ we say that u and v are respectively a *prefix* and a *suffix* of w. The suffix v is said to be *proper* if $v \neq w$.

For a finite set X of words on A, we denote by $\mathit{Pref}(X)$ and $\mathit{Suff}(X)$ the set of prefixes and suffixes of the words of X respectively, by $\mathrm{Card}(X)$ the

cardinality of X and by $|X|$ the sum of the lengths of words of X, that is

$$|X| = \sum_{x \in X} |x|.$$

We denote a nondeterministic finite automaton over the alphabet A by $\mathcal{A} = (Q, \delta, i, T)$ where Q is the set of states, $i \in Q$ is the initial state, $T \subseteq Q$ is the set of terminal states and δ is the transition function. We use $\sharp\mathcal{A}$ to denote the number of states of $\mathcal{A}$. We abbreviate by NFA a nondeterministic finite automaton and by DFA a deterministic finite automaton.

3 Using deterministic automata

Before presenting our algorithm, we examine what would be the classical approach to check if a word w is in X^*. A natural idea is to build an automaton for X^*. We consider the following process: (i) build a finite automaton for X, (ii) modify this automaton to accept X^* (doing so, we usually get an NFA), and (iii) finally get a DFA after a classical determinization procedure. An optional fourth step could be to minimize the resulting automaton.

Automata for a finite set of words X. First we consider three simple ways of building an automaton for a finite set of words X.

(1) The "solar" automaton $\mathcal{S}_X$ which is obtained as follows: we build one automaton per word $x \in X$ with $|x| + 1$ states and merge all the initial states (see Fig. 1). Note that this NFA is a tree with root i and that $\sharp\mathcal{S}_X = |X| + 1$.

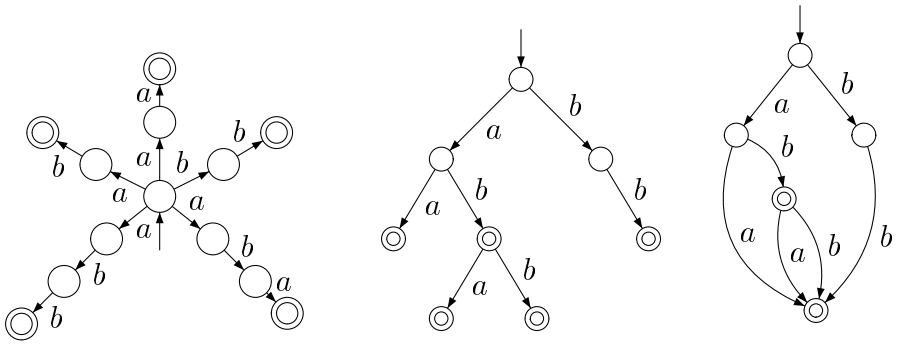

Fig. 1. The solar NFA $\mathcal{S}_X$ (left), the tree DFA $\mathcal{T}_X$ (middle) and the minimal DFA $\mathcal{M}_X$ for $X = \{aa, ab, bb, aba, abb\}$.

(2) The tree automaton $\mathcal{T}_X$ (see Fig. 1): this is a tree which collects words sharing a common prefix. In terms of automata, the set of states corresponds to the set of prefixes and we have

$$\mathcal{T}_X = (Q = \mathit{Pref}(X), \delta, i = \varepsilon, T = X)$$

with $\delta(p, a) = pa$ if $p, pa \in \mathit{Pref}(X)$ and $a \in A$. This DFA has $\sharp\mathcal{T}_X = \mathrm{Card}(\mathit{Pref}(X))$ states.

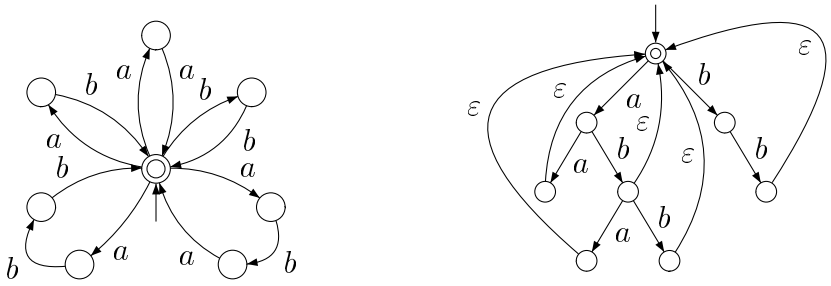

Fig. 2. The flower automaton $\mathit{merge}(\mathcal{S}_X)$ (left) and the NFA $\mathit{star}(\mathcal{T}_X)$ (right) for $X = \{aa, ab, bb, aba, abb\}$.

(3) The minimal automaton $\mathcal{M}_X$ (see Fig. 1). Given the set X a more elaborate method is to build the minimal DFA $\mathcal{M}_X$ recognizing X. For instance one can apply a minimization algorithm to the tree automata $\mathcal{T}_X$ in *linear time* with respect to $|X|$ [12]. Of course $\mathcal{M}_X$ is not necessary a tree.

Automata for the language X^*. A straightforward way to build an NFA recognizing the language X^* from an automaton $\mathcal{A} = (Q, \delta, i, T)$ recognizing X is to add ε-transitions from each final state of T to the initial state i. We denote this automaton by $\mathit{star}(\mathcal{A})$ (see Fig. 2). To save a little more space, we also merge all the terminal states without outgoing transitions with the initial state: this yields an automaton $\mathit{merge}(\mathcal{A})$. Doing so with the solar automaton $\mathcal{S}_X$, we obtain $\mathit{merge}(\mathcal{S}_X)$ the classical flower automaton of X^* (see Fig. 2).

Applying the classical powerset construction to one of the previous automata accepting X^*, we obtain a DFA for X^*.

Note that the determinization procedure gives the same result either starting from $\mathit{star}(\mathcal{S}_X)$ or from $\mathit{star}(\mathcal{T}_X)$, due to their tree-like structures. The same is true with *merge* instead of *star*.

In general, for an NFA $\mathcal{A}$, the determinization procedure builds a DFA whose number of states is trivially bounded by $2^{\sharp\mathcal{A}}$. However, when we consider the particular case of X^* with X finite, could an exponential blow-up really happen? The following example shows that the answer is positive.

Example 1 *Let us consider $X = A^k a + b$ with $k > 0$. It is easy to give an NFA for X^* with $k+1$ states (see Fig. 3). The determinization leads to a DFA with $\Theta(2^k)$ states.*

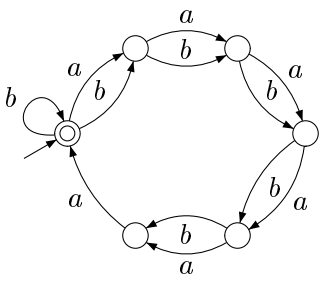

Fig. 3. An NFA for X^* with the set $X = A^k a + b$ ($k = 5$) of Example 1.

Another question is to find if we can relate the number of states of a DFA for X^* to $|X|$. Until recently, it was thought that it could not exceed $\Theta(|X|^2)$, a bound which was shown to be reachable in [13], as stated in the following example.

Example 2 *For an integer $h > 1$, take $X = \{a^{h-1}, a^h\}$. The tree DFA $\mathcal{T}_X$ and the minimal DFA $\mathcal{M}_X$ are the same and have $h+1 = \Theta(|X|)$ states. The minimal DFA for X^* has $\Theta(|X|^2)$ states (see [13]).*

J. Shallit showed in [14] with the following example that an exponential blow-up is also possible.

Example 3 *Let $h \geq 3$ and let*

$$X = \{b\} \cup \{a^i b a^{h-i-1} | 1 \leq i \leq h-2\} \cup \{b a^{h-2} b\}.$$

The minimal DFA accepting X^* has exactly $3(h-1)2^{h-3} + 2^{h-2}$ states [14]. Note that the size is exponential of order $\Theta(h2^h)$ whereas $\mathrm{Card}(X) = \Theta(h)$ and $|X| = \Theta(h^2)$.

The problem of finding a tight upper bound for the number of states of the minimal DFA for X^* in terms of the total length $|X|$ is called by this author [14] the *non-commutative Frobenius problem.*

The number of states of the minimal automata obtained for the family of sets used in Example 3 is $\Theta(h2^h)$ where $h = \Theta(|X|^{1/2})$. A priori, the upper bound for a DFA obtained by determinization of a NFA for X^* with $\theta(|X|)$ states is $\Theta(2^{|X|})$. Experiments performed on the family of Example 3 show that the DFA obtained by determinization (before minimization) has also $\Theta(h2^h)$ states, and not $\Theta(2^{h^2})$. We do not know in general whether

(i) it is possible that the minimal DFA for X^* has $\Theta(2^{|X|})$ states;

(ii) it is possible that the DFA obtained by determinization has $\Theta(2^{|X|})$ states.

Simulating the determinization process. A way to avoid the determinization step would be to simulate the determinized automaton while parsing the word w. Given a NFA $\mathcal{A}$ accepting the language X^* with X a finite set of words, this gives an algorithm of time complexity $O(\sharp\mathcal{A} \times |w|)$ and the space required to simulate the determinization process is $\sharp\mathcal{A}$. Since the number of states of the NFA can be of order $O(|X|)$, this approach gives a time complexity $O(|X| \times |w|)$ in the worst case. As an example of such a situation, we have the set $X = \{a^k b, a\}$ with $k > 0$.

4 Using string matching machines

The methods discussed in the previous section do not lead to an optimal algorithm in $O(|w|)$. Indeed, either we use a DFA and we face a computation which can be exponential in $|X|$ or we simulate the DFA and we obtain an algorithm in $O(|X| \times |w|)$. We now consider a different approach which leads to a lower complexity.

Another advantage of the proposed approach is to possibly solve a more general problem. Indeed, we may be interested in obtaining the set of all decompositions of the input word over X. This cannot be achieved using a DFA accepting X^* given for instance by the methods described in the previous section.

Let $X = \{x_0, \ldots, x_{n-1}\}$ be a set of n words of A^+. We present in this section an algorithm, using classic pattern matching techniques, which gives all the *X-decompositions* of w (the decompositions of w as concatenations of words of X). Then we derive a membership test for X^* in $O(\mathrm{Card}(X) \times |w|)$ time complexity. In the next section, we shall study a further improvement of this algorithm.

4.1 Decompositions

The following remark is the basis of our algorithm. An X-decomposition of w is always the extension of an X-decomposition of a prefix of w.

We consider the prefix $w[0..i]$ of length $i+1$ of w. The word $w[0..i]$ admits an X-decomposition ending with a word x_ℓ if and only if $w[0..i] = fx_\ell$ for a word f in X^*. In other terms, $w[0..i]$ admits an X-decomposition ending with x_ℓ if and only if x_ℓ is a suffix of $w[0..i]$ and $w[0..i - |x_\ell|] \in X^*$. We obtain all

the X-decompositions of $w[0..i]$ by examining all the words of X which are suffixes of $w[0..i]$ and which extend a previous X-decomposition. Of course, when $w[0..i] = w$, we get all the X-decompositions of w.

So the idea of the algorithm is the following: build, for each word $x_\ell \in X$, a deterministic automaton $\mathcal{A}_\ell$ accepting the language A^*x_ℓ and use an auxiliary array D of size $|w|$ such that

$$D[i] = \{\ell \in [0..n-1] \mid w[0..i] \in X^*x_\ell\}.$$

Then testing if $w[0..i]$ ends by the word x_ℓ is equivalent to checking that the automaton $\mathcal{A}_\ell$ is in a terminal state after reading $w[0..i]$. Also testing if $w[0..i-|x_\ell|] \in X^*$ is equivalent to checking that $D[i-|x_\ell|] \neq \emptyset$.

In the following algorithm, the input word w is read simultaneously by all the n automata, letter by letter, from left to right. We use, for technical convenience, an additional element $D[-1]$ initialized to an arbitrary non-empty set (for instance $\{\infty\}$) meaning that the prefix ε of w is always in X^*. At the end of the scanning of w, provided $D[|w|-1] \neq \emptyset$, we can process the array D from the end to the beginning and recover all the X-decompositions for instance with a recursive procedure like PrintAllDecompositions() (see below).

For each word $x_\ell \in X$, the automaton $\mathcal{A}_\ell$ considered here is the minimal automaton which recognizes the language A^*x_ℓ. This automaton is defined by $\mathcal{A}_\ell = (Q_\ell = \mathit{Pref}(x_\ell), \delta_\ell, i_\ell = \varepsilon, t_\ell = x_\ell)$ where the transition function δ_ℓ is defined, for $p \in \mathit{Pref}(X)$ and $a \in A$, by

$$\delta_\ell(p,a) = \text{the longest suffix of } pa \text{ which belongs to } \mathit{Pref}(x_\ell).$$

We use these principles in the following algorithm.

```
IsDecomposedAll(w, X = {x_0, ..., x_{n-1}})
 1  ▷ Preprocessing step
 2  for ℓ ← 0 to n − 1 do
 3        A_ℓ ← AutomatonFromWord(x_ℓ)
 4  ▷ Main loop
 5  for ℓ ← 0 to n − 1 do
 6        ▷ p_ℓ is the current state of the automaton A_ℓ.
 7        p_ℓ ← i_ℓ
 8  D[−1] ← {∞}
 9  for i ← 0 to |w| − 1 do
10        D[i] ← ∅
11        for ℓ ← 0 to n − 1 do
12              p_ℓ ← δ_ℓ(p_ℓ, w[i])
13              if p_ℓ = t_ℓ and D[i − |x_ℓ|] ≠ ∅ then
14                    D[i] ← D[i] ∪ {ℓ}
15  return D
```

The algorithm returns an array of size $O(\mathrm{Card}(X) \times |w|)$. The preprocessing step which builds automata requires a time $O(|X|)$ and a space $O(|X| \times \mathrm{Card}(A))$ (or $O(|X|)$ if automata are represented with the help of a failure function as usually made in stringology [15,16]).

Note that we do not need to build all the automata $\mathcal{A}_\ell$ in the preprocessing step. We can also choose to construct in a lazy way the accessible part of the automata (corresponding for each automaton $\mathcal{A}_\ell$ to the prefixes of x_ℓ occurring in w) along the processing of the input word w. For the sake of clarity, we have chosen to distinguish the preprocessing step from the rest. In view of this remark, we could omit the complexity $O(|X|)$ of the preprocessing step in the following proposition.

Proposition 4 *The time and space complexity of the algorithm* IsDecomposedAll() *is* $O(\mathrm{Card}(X) \times |w| + |X|)$.

Given the array D computed by the procedure IsDecomposedAll() for a word w, it is quite straightforward to print all the decompositions of w thanks to the following two procedures:

```
PrintAllDecompositions(w, X = {x_0, ..., x_{n-1}})
 1  D ← IsDecomposedAll(w, X)
 2  L ← emptyList
 3  RecPrintAllDecompositions(D, |w| − 1, L)
```

```
RecPrintAllDecompositions(D, h, L)
 1  if h = −1 then
 2        Print(L)
 3  else for j ∈ D[h] do
 4              RecPrintAllDecompositions(D, h − |x_j|, x_j · L)
```

For a word w belonging to X^* the procedure PrintAllDecompositions() prints every X-decomposition of w in the form $x_{i_0} \cdot x_{i_1} \cdots x_{i_p}$.

If we want only one X-decomposition of w, it suffices to store in $D[i]$ only one word x of X corresponding to an X-decomposition of $w[0..i]$ ending with this x. The space required for the array then becomes $O(|w|)$.

4.2 Membership test

When we are only interested in testing the membership of w in X^*, we can simply use a Boolean array D setting $D[i] =$ true if and only if there exists

$x \in X$ such that $w[0..i] \in X^*x$. Moreover, it suffices to use a circular Boolean array $D[0..k]$ with $k = \max_{x \in X} |x|$ (instead of $|w|+1$), and compute indexes in this array modulo $k+1$ (which means that for $m \in \mathbb{Z}$, one has $D[m] = D[r]$ with $0 \leq r < k+1$ and $m = r \bmod (k+1)$). This yields the following algorithm.

```
MEMBERSHIP(w, X = {x_0, ..., x_{n-1}})
 1  ▷ Preprocessing step
 2  for ℓ ← 0 to n − 1 do
 3        A_ℓ ← AUTOMATONFROMWORD(x_ℓ)
 4  ▷ Main loop
 5  for ℓ ← 0 to n − 1 do
 6        ▷ p_ℓ is the current state of the automaton A_ℓ.
 7        p_ℓ ← i_ℓ
 8  D[−1] ← true
 9  for i ← 0 to |w| − 1 do
10        D[i] ← false
11        for ℓ ← 0 to n − 1 do
12              p_ℓ ← δ_ℓ(p_ℓ, w[i])
13        ℓ ← 0
14        do  if (p_ℓ = t_ℓ and D[i − |x_ℓ|] = true) then
15                  D[i] ← true
16              ℓ ← ℓ + 1
17        while (ℓ < n and D[i] = false)
18  return D[|w| − 1]
```

We can easily modify the algorithm while preserving the same complexity by exiting whenever all the elements of the array D from 0 to k are equal to false. In this case, $w \notin X^*$.

The following proposition gives the complexity of the above algorithm.

Proposition 5 *The time complexity of the algorithm* MEMBERSHIP() *is* $O(\mathrm{Card}(X) \times |w| + |X|)$.

The analysis of the space complexity shows that, except for the preprocessing step, the algorithm needs only $O(\max_{x \in X} |x|)$ additional space. In particular, the space complexity is independent of the length of the input word.

5 String matching automaton

In the preceding section, we used for each word $x_\ell \in X$ a distinct automaton $\mathcal{A}_\ell$ corresponding to A^*x_ℓ. To get a more efficient algorithm, we resort in this section to the well-known Aho-Corasick algorithm [17] which builds from a

finite set of words X a deterministic complete automaton (not necessarily minimal) $\mathcal{A}_X$ recognizing the language A^*X. This automaton is the basis of many efficient algorithms on string matching problems and is often called the *string matching automaton*. It is a generalization of the automaton $\mathcal{A}_\ell$ associated to a single word. Let us briefly recall its construction. We let $\mathcal{A}_X = (\mathit{Pref}(X), \delta, \varepsilon, \mathit{Pref}(X) \cap A^*X)$ be the automaton where the set of states is $\mathit{Pref}(X)$, the initial state is ε, the set of final states is $\mathit{Pref}(X) \cap A^*X$ and the transition function δ is defined by

$$\delta(p, a) = \text{the longest suffix of } pa \text{ which belongs to } \mathit{Pref}(X).$$

We associate to each word $u \in A^*$, $u \neq \varepsilon$, the word $\mathit{Border}_X(u)$, or simply $\mathit{Border}(u)$ when there is no ambiguity, defined by

$$\mathit{Border}(u) = \text{the longest proper suffix of } u \text{ which belongs to } \mathit{Pref}(X).$$

The automaton $\mathcal{A}_X$ can be easily built from the tree $\mathcal{T}_X$ (cf. the section 3) of X by a breadth-first exploration and using the Border function. Indeed, one has

$$\delta(p, a) = \begin{cases} pa & \text{if } pa \in \mathit{Pref}(X) \\ \delta(\mathit{Border}(p), a) & \text{if } p \neq \varepsilon \text{ and } pa \notin \mathit{Pref}(X) \\ \varepsilon & \text{otherwise} \end{cases}$$

A state p is terminal for $\mathcal{A}_X$ if p is a word of X (i.e. p is terminal in the tree $\mathcal{T}_X$ of X) or if a proper suffix of p is a word of X. The automaton $\mathcal{A}_X$ can be built in time and space complexity $O(|X|)$ if we use the function Border as a failure function (see [15,16] for implementation details).

We will say, for simplicity, that a state of the automaton is *marked* if it corresponds to a word of X and *not marked* otherwise. A major difference induced by the Aho-Corasick automaton is that a terminal state p, marked or not, corresponds to an ordered set $\mathit{Suff}(p) \cap X$ of suffixes of p. The order considered is given by the suffix relation $\succ_{\text{suff}}$ where $u \succ_{\text{suff}} v$ means that v is a proper suffix of u. We denote by $\mathit{SuffixChain}(p)$ the sequence of words in $\mathit{Suff}(p) \cap X$ ordered by this relation. To find easily the words of $\mathit{SuffixChain}(p)$, we associate to each terminal state p of $\mathcal{A}_X$ the state

$$\mathit{SuffixLink}(p) = \text{the longest proper suffix of } p \text{ which belongs to } X.$$

Thus we have

$$\mathit{SuffixLink}(p) = \begin{cases} \mathit{Border}(p) & \text{if } \mathit{Border}(p) \in X \\ \mathit{SuffixLink}(\mathit{Border}(p)) & \text{if } \mathit{Border}(p) \notin X \text{ and } \mathit{Border}(p) \neq \varepsilon \\ \text{undefined} & \text{otherwise} \end{cases}$$

Since *SuffixLink*(p) is computed in time $O(|p|)$, the preprocessing can be done in time and space complexity $O(|X|)$, i.e. the complexity of Aho-Corasick algorithm.

To decide whether an input word w belongs to X^* or not (and get eventually its X-decompositions), we use the same technique as in the previous section, considering this time the automaton $\mathcal{A}_X$ (instead of the n automata $\mathcal{A}_\ell$). The immediate advantage is that each letter of the word w is read only once (meaning that only one transition is made in the automaton) whereas each letter was read n times before (one per automaton $\mathcal{A}_\ell$).

Let us suppose that for the current prefix $w[0..i]$ of w, the automaton $\mathcal{A}_X$ ends in a terminal state p. This means that $w[0..i] = fp$ with $f \in A^*$ and p the longest suffix of $w[0..i]$ in $\mathit{Pref}(X) \cap A^*X$. Consequently, $w[0..i] \in X^*$ if and only if $w[0..i-|x|] \in X^*$ for at least one word x of *SuffixChain*(p). This is easily checked using the marking of terminal states (whether they correspond exactly to a word of X or not), the function *SuffixLink*(p) and the array D (which plays exactly the same role as in the previous section).

This yields our main result, stated in the following proposition.

Proposition 6 *Let X be a finite set of words on A. The membership test of a word w in X^* can be done in time $O(r \times |w| + |X|)$ where r is the maximal length of the suffix chains in X.*

The space complexity includes $O(|X|)$ for the preprocessing step (building the Aho-Corasick automaton) and $O(\max_{x\in X} |x|)$ for the rest of the algorithm.

If X is a suffix code, the complexity, except for the preprocessing step, becomes $O(|w|)$ which is optimal, whereas the worst case happens when all words are suffixes of one another giving the same complexity $O(\mathrm{Card}(X) \times |w|)$ as in the previous section. Note also that in the particular case where X is a prefix code, it is easy to solve the membership problem for X^* in an optimal time $O(|w|)$ after a $O(|X|)$ preprocessing step.

Example 7 *Let $X = \{a^2, a^4b, a^3ba, a^2b, ab\}$. For the word $w = a^5b$, it is necessary to follow the suffix chain $\mathit{SuffixChain}(a^4b) = (a^4b, a^2b, ab)$ since after parsing w the automaton is in the state corresponding to a^4b and the unique X-decomposition is $a^5b = a^2 \cdot a^2 \cdot ab$. Figure 4 shows the tree $\mathcal{T}_X$ (left), the automaton $\mathcal{A}_X$ with the links representing the failure function Border (middle) and the SuffixLink representing the suffix chains (right) to add to the Aho-Corasick automaton.*

Acknowledgements We thank the referee for pointing us the reference to J. Shallit's [14] used in Example 3. The style for algorithms is algochl.sty from

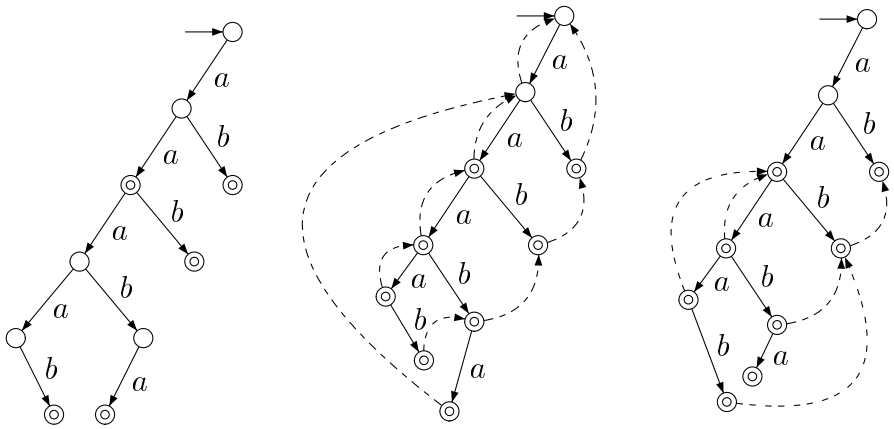

Fig. 4. For the set $X = \{a^2, a^4b, a^3ba, a^2b, ab\}$ of example 7: Tree $\mathcal{T}_X$ (left), Aho-Corasick automaton with links *Border* (middle) and the new links *SuffixLink* (right) to add to the Aho-Corasick automaton.

[16] and automata are drawn thanks to gastex.

References

[1] M.-P. Schützenberger, A property of finitely generated submonoids of free monoids, in: G. Pollak (Ed.), Algebraic Theory of Semigroups (Proc. Sixth Algebraic Conf., Szeged,1976), North-Holland, Amsterdam, 1979, pp. 545–576.

[2] A. Sardinas, G. Patterson, A necessary and sufficient condition for the unique decomposition of coded messages, in: IRE Convention Record, Part 8, 1953, pp. 104–108.

[3] A. Apostolico, R. Giancarlo, Pattern matching implementation of a fast test for unique decipherability, Information Processing Letters 18 (1984) 155–158.

[4] M. Rodeh, A fast test for unique decipherability based on suffix trees, IEEE Trans. Inform. Theory 28 (1982) 648–651.

[5] C. M. Hoffmann, A note on unique decipherability, in: MFCS, Vol. 176 of Lecture Notes in Computer Science, Springer-Verlag New York, Inc., 1984, pp. 50–63.

[6] R. McCloskey, An $o(n^2)$ time algorithm for deciding whether a regular language is a code, Journal of Computing and Information 2 (1) (1996) 79–89, special Issue: *Proceedings of the 8th international Conference on Computing and Information (ICCI'96).*

[7] Z. Galil, Open problems in stringology, in: A. Apostolico, Z. Galil (Eds.), Combinatorial Algorithms on Words, Springer-Verlag, Berlin, 1985, pp. 1–8.

[8] D. Perrin, Finite automata, in: J. Leeuwen (Ed.), Handbook of Theoretical Computer Science, Formal Models and Semantics, Vol. B, Elsevier, 1990, pp. 1–57.

[9] J. Hopcroft, R. Motwani, J. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 2001.

[10] J. Sakarovitch, Eléments de théorie des automates, Vuibert, 2003.

[11] S. Yu, Regular languages, in: G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, Springer-Verlag New York, Inc., 1997, pp. 41–110.

[12] D. Revuz, Minimisation of acyclic deterministic automata in linear time, Theor. Comput. Sci. 92 (1) (1992) 181–189.

[13] S. Yu, State complexity of regular languages, in: Proceedings of Descriptional Complexity of Automata, Grammars and Related Structures, 1999, pp. 77–88.

[14] J. Shallit, Regular expressions, enumeration and state complexity, invited talk at Ninth International Conference on Implementation and Application of Automata (CIAA 2004) Queen's University, Kingston, Ontario, Canada, July 22-24, 2004.

[15] M. Crochemore, W. Rytter, Jewels of Stringology, World Scientific Publishing, Hong-Kong, 2002, 310 pages.

[16] M. Crochemore, C. Hancart, T. Lecroq, Algorithmique du texte, Vuibert, 2001, 347 pages.

[17] A. V. Aho, M. J. Corasick, Efficient string matching: an aid to bibliographic search, Commun. ACM 18 (6) (1975) 333–340.