# Efficient Experimental String Matching by Weak Factor Recognition*

Cyril Allauzen[1] Maxime Crochemore[1] Mathieu Raffinot[2]

[1] Institut Gaspard-Monge, Université de Marne-la-Vallée, Cité Descartes, Champs-sur-Marne, 77454 Marne-la-Vallée Cedex 2, France, E-mail: {allauzen,mac}@monge.univ-mlv.fr

[2] Equipe génome, cellule et informatique, Université de Versailles, 45 avenue des Etats-Unis, 78035 Versailles Cedex, France, E-mail: raffinot@genome.uvsq.fr

**Abstract.** We introduce a new notion of *weak factor recognition* that is the foundation of new data structures and *on-line* string matching algorithms. We define a new automaton built on a string $p = p_1p_2 \ldots p_m$ that acts like an oracle on the set of factors $p_i \ldots p_j$. If a string is recognized by this automaton, it may be a factor of $p$. But, if it is rejected, it is surely not a factor. We call it *factor oracle*. More precisely, this automaton is acyclic, recognizes at least the factors of $p$, has $m + 1$ states and a linear number of transitions. We give a very simple sequential construction algorithm to build it. Using this automaton, we design an efficient experimental *on-line* string matching algorithm (we conjecture its optimality in regard to the experimental results) that is really simple to implement. We also extend the *factor oracle* to predict that a string could be a suffix (*i.e.* in the set $p_i \ldots p_m$) of $p$. We obtain the *suffix oracle*, that enables in some cases a tricky improvement of the previous string matching algorithm.

**Keywords**: finite automaton, string matching, algorithm design, information retrieval.

## 1 Introduction

A *string* $p$ is a sequence $p = p_1p_2 \ldots p_m$ of letters taken in a finite alphabet $\Sigma$. We keep the notation $p$ along this paper to denote the string we are working on. A factor of $p$ is a string $p_i \ldots p_j$, $1 \le i \le j \le m$.

The basic string matching problem is to find all occurrences of a pattern string $p$ in a large text $T$. Efficient *on-line* string matching algorithms are based on indexes built on $p$.

Many indexing techniques exist for this purpose. The simplest methods use precomputed tables of $q$-grams while more advanced methods use more elaborated data structures. These classical structures are: suffix arrays, suffix trees, suffix automata or DAWGs, and factor automata (see [8] for a survey).

* Work partially supported by Wellcome Trust Foundation and by NATO Grant PST.CLG.977017.

The notion on which these structures and these *on-line* string matching algorithms are based is *exact factor recognition.* It means that we need to know if a given string $u$ is or is not a factor of the pattern $p$. This notion leads to very time-efficient string matching algorithms, but presents two major drawbacks: (a) the structures require a fairly large amount of memory space (which implies a large number of memory page breaks when searching for the string in the text); (b) the algorithms are rather involved to implement.

It is considered, for example, that the implementation of suffix arrays can be achieved using five bytes per string character and that other structures need about twelve bytes per string character.

We propose in this paper a new approach based on the notion of *weak factor recognition.* The idea is to recognize more than the exact factors of $p$ to win in simplicity and memory requirements.

For this purpose, we build a new structure, called *factor oracle*, that can replace many of these indexes in *on-line* string matching algorithm. More precisely, this structure is an automaton (a) that is acyclic (b) that recognizes at least the factors of $p$ (c) that has the fewest states possible (*i.e.* $m+1$) and (d) that has a linear number of transitions. The suffix and factor automata [5, 7] satisfy (a)-(b)-(d) but not (c) whereas the sub-sequence automaton [4] satisfies (a)-(b)-(c) but not (d).

We give two different construction algorithms for the factor oracle, the first is only conceptual (and is used as definition), the second is a really simple practical sequential algorithm.

The relations between our factor oracle and the suffix automaton allow us to define another new structure : the *suffix oracle.*

From a theoretical point of view, these two structures are of interest. They represent the first attempt to formalize the notion of *weak factor recognition.* Although their constructions are very simple, their properties are rather difficult to establish and many points remain open and require further studies.

We use these two new structures to design new experimental *on-line* string matching algorithms. These algorithms have a very good average behavior that we conjecture optimal. The main advantages of these new algorithms are (1) that they are easy to implement for an optimal behavior and (2) that they are in pratice as fast as the fastest ones. A preliminary abstract version of this paper appears in [2].

The factor oracle can be extended to a set of strings, and be used in multi string matching algorithms, that leads to very promising experimental results [3].

We now define the notions and the notations we need along this paper. We denote $\mathrm{Fact}(p)$ the set of all the factors of string $p$. A factor $x$ of $p$ is a *prefix* (resp. a *suffix*) of $p$ if $p = xu$ (resp. $p = ux$) with $u \in \Sigma^*$. The set of all the prefixes of $p$ is denoted by $\mathrm{Pref}(p)$ and the one of all the suffixes $\mathrm{Suff}(p)$. We say that $x$ is a *proper factor* (resp. *proper prefix, proper suffix*) of $p$ if $x$ is a factor (resp. prefix, suffix) of $p$ distinct from $p$ and from the empty string $\epsilon$. We denote $\mathrm{pref}_p(i)$ the prefix of length $i$ of $p$ for $0 \le i \le |p|$. We denote for $u \in \mathrm{Fact}(p)$, $\mathrm{poccur}(u,p) = \min\{|z|\ ,\ z = wu \text{ and } p = wuv\}$, the ending position of the first occurrence of $u$ in $p$. Finally, we define for $u \in \mathrm{Fact}(p)$ the set $\mathrm{endpos}_p(u) = \{i \mid p = wup_{i+1} \ldots p_m\}$.

## 2 Factor oracle

### 2.1 Construction algorithm

**Build_Oracle**($p = p_1 p_2 \ldots p_m$)
For $i$ from 0 to $m$
    Create a new state $i$
For $i$ from 0 to $m-1$
    Build a new transition from $i$ to $i+1$ by $p_{i+1}$
For $i$ from 0 to $m-1$
    Let $u$ be a minimal length word in state $i$
    For all $\sigma \in \Sigma, \sigma \neq p_{i+1}$
    If $u\sigma \in \mathrm{Fact}(p_{i-|u|+1} \ldots p_m)$
    Build a new transition from $i$ to $i + \mathrm{poccur}(u\sigma, p_{i-|u|+1} \ldots p_m)$ by $\sigma$

**Fig. 1.** High-level construction algorithm of Oracle($p$).

**Definition 1** *The* factor oracle *of a string $p = p_1 p_2 \ldots p_m$ is the automaton build by the algorithm* Build_Oracle *(Figure 1) on the string $p$, where all the states are terminal. It is denoted by Oracle($p$).*

A string $w$ is recognized in state $i$ by the factor oracle if it labels a path from state 0 to state $i$. The factor oracle of the string $p = abbbaab$ is given as an example in Figure 2. On this example, it can be noticed that the string $aba$ is recognized whereas it is not a factor of $p$.

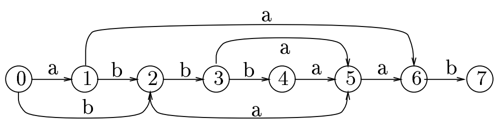



**Fig. 2.** Factor oracle of $abbbaab$. The word $aba$ is recognized whereas it is not a factor.

Note: all the transitions that reach state $i$ of Oracle($p$) are labeled by $p_i$.

**Lemma 1** *Let $u \in \Sigma^*$ be a minimal length string among the strings recognized in state $i$ of Oracle($p$). Then, $u \in$ Fact($p$) and $i =$ poccur($u, p$).*

*Proof.* By induction on the number of the state $i$. It is true for states 0 and 1. Assume that it is true for all states $0 \leq j \leq i - 1$. Let $u$ be a minimal length string among the strings recognized in state $i$. We consider the last transition on the path labeled by $u$ leading from 0 to $i$.

(i) This transition was built at line 2. Then $u = zp_i$ with $z \in \Sigma^*$. As $u$ is a minimal length string of state $i$, $z$ is a minimal length string for state $i - 1$. By the induction hypothesis, $z \in \text{Fact}(p)$ and $i - 1 = \text{poccur}(z, p)$. And therefore, $u = zp_i \in \text{Fact}(p)$ and $i = \text{poccur}(u, p)$.
(ii) This transition was built at line 7. Then it leads from $j$ to $i$ labeled by $p_i$ with $0 < j < i - 1$. So $u = zp_i$ and $z$ is a minimal length string for state $j$. By the induction hypothesis, $z \in \text{Fact}(p)$ and $j = \text{poccur}(z, p)$. And on account of the construction of the transition line 7, $u = zp_i \in \text{Fact}(p)$ and $i = \text{poccur}(u, p)$.

□

**Corollary 1** *Let $u \in \Sigma^*$ be a minimal length string among the strings recognized in state $i$ of Oracle($p$), $u$ is unique.*

We denote $\min(i)$ the minimal length string recognized in state $i$.

**Corollary 2** *Let $i$ and $j$ be two states of Oracle($p$) such as $j < i$. Let $u = min(i)$ and $v = min(j)$, $u$ can not be a suffix of $v$.*

*Proof.* Assume that $u$ is a suffix of $v$. In this case, $\text{poccur}(u, p) \leq \text{poccur}(v, p)$ which is a contradiction of (according to lemma 1) $j < i$. □

**Lemma 2** *Let $i$ be a state of Oracle($p$) and $u = min(i)$. $u$ is a suffix of any string $c \in \Sigma^*$ recognized in state $i$.*

*Proof.* By induction on the number of the state $i$. It is true for states 0 and 1. Assume that it is true for all states $0 \leq j \leq i - 1$. We consider state $i$ and let $u = \min(i)$.

Let $c$ be a path leading to state $i$, $c = c_1 a$ where $c_1$ leads to state $j < i$. Let $v = \min(j)$. Then

- $|va| \geq |u|$ because $u$ is the minimum of $i$ and $va$ leads to $i$.
- according to the construction (Figure 1 line 7), $i \in \text{endpos}_p(va)$ because $v = min(j)$ and there is a transition from $j$ to $i$ by $a$.
- lemma 1 sets $i \in \text{endpos}_p(u)$.

From these three facts, it comes that $u$ is a suffix of $va$. As $j < i$, by the induction hypothesis, $v$ is a suffix of $c_1$ and therefore $u$ is a suffix of $c$. □

**Lemma 3** *Let $w \in Fact(p)$. $w$ is recognized by Oracle($p$) in a state $j \leq poccur(w,p)$.*

*Proof.* By induction on the length of $w = w_0 w_1 \dots w_f$. We denote $i = \mathrm{poccur}(w,p)$.

- There is a transition from 0 by $w_0$ leading to a state $k_0 \leq i - f$.
- Assume that there is a path labeled by $w_0 \dots w_j$ leading to a state $k_j \leq i - f + j$. Let $u = \min(k_j)$. According to lemma 2, $u = w_{j-|u|+1} \dots w_j$. As $uw_{j+1}$ is a factor of $p$ and as $i - f + (j+1) \in \mathrm{endpos}_p(uw_{j+1})$, there is a transition from $k_j$ labeled $w_{j+1}$ leading to a state $k_{j+1} \leq i - f + (j+1)$.

□

**Corollary 3** *Let $w \in Fact(p)$. Every string $v \in Suff(w)$ is recognized by Oracle($p$) in a state $j \leq poccur(w)$.*

**Lemma 4** *Let $i$ be a state of Oracle($p$) and $u = min(i)$. Any path ending by $u$ leads to a state $j \geq i$.*

*Proof.* By induction on the number of the state $i$. It is true for states 0 and 1. Assume that it is true for all states $0 \leq j \leq i-1$. We consider state $i$. Let $u$ be a minimal length string among the strings recognized in state $i$.

Consider the minimal length path labeled by $u$ leading to $i$. We denote $j$ the state preceding $i$ along this path, and $v = \min(j)$. We have $u = va$ by construction. Consider a path $c = c_1 u$ leading to a state $k$. Assume that $k < i$.

By the induction hypothesis, the path $c_1 v$ leads to a state $l \geq j$. If $l = j$, then $k = i$ which is a contradiction with $k < i$. We have now $j < l < k < i$. Let $w = \min(l)$.

- If $|w| > |v|$, then $v$ is a suffix of $w$ (lemma 2). In that case, $u = va$ is a suffix of $wa$, so that $k \in \mathrm{endpos}_p(u)$ which is a contradiction with $k < i = \mathrm{poccur}(u,p)$ (lemma 1).
- If $|w| \leq |v|$, then $w$ is a suffix of $v$ (lemma 2). This is a a contradiction with $j < l$ (corollary 2).

In both cases, we reach a contradiction, so that $k \geq i$ and the induction hypothesis is verified for $i$. □

**Lemma 5** *Let $w \in \Sigma^*$ be a string recognized by Oracle($p$) in $i$, then any suffix of $w$ is recognized in a state $j \leq i$.*

*Proof.* By induction on $|w|$. It is true if $|w| = 0$ or $|w| = 1$. Assume that it is true for all the strings $\zeta$ such that $|\zeta| < |w|$. We show that it is also true for $w$, recognized in $i$.

Let $w = \zeta a$, $\zeta$ is recognized in $k < i$. Consider a proper suffix of $w$. It can be written $va$ where $v$ is a proper suffix of $\zeta$.

According to the induction hypothesis, $v$ is recognized in $l \leq k$. Let $\bar{\zeta} = min(k)$ and $\bar{v} = min(l)$. The lemma 2 implies that $\bar{\zeta}$ is a suffix of $\zeta$ and that $\bar{v}$ is a proper suffix of $v$. The corollary 2 sets that $\bar{v}$ is a suffix of $\bar{\zeta}$. As $i \in \mathrm{endpos}_p(\bar{\zeta}a)$ (by construction of the transition by $a$), $i \in \mathrm{endpos}_p(\bar{v}a)$. So here is a transition from $l$ by $a$ leading to a state $j \leq i$. Thus, the proper suffix $va$ is recognized in state $j$. □

The number of states of Oracle($p$) with $p = p_1p_2 \ldots p_m$ is $m + 1$. We now consider the number of transitions.

**Lemma 6** *The number $T_{Or}(p)$ of transitions in Oracle($p = p_1p_2 \ldots p_m$) satisfies $m \leq T_{Or}(p) \leq 2m - 1$.*

*Proof.* There is always $m$ transitions $i \to i + 1$ labeled by $p_{i+1}$. As the strings $a^m$ only have these transitions, $m$ is the minimal number of transitions.

Consider now the transitions $i \to j$ where $j > i+1$. We build an injective function that maps each of these transitions to a proper suffix of $p$. Each transition $i \to j$ with $j > i + 1$ labeled by $\sigma$ is mapped to the string $\min(i)\sigma p_{j+1} \ldots p_m$. By construction of Oracle($p$), this string is a proper suffix of $p$. We show *a contrario* that this function is injective. Assume that there are two distinct transitions $i_1 \to j_1$ and $i_2 \to j_2$ respectively labeled by $\sigma_1$ and $\sigma_2$ such that $\min(i_1)\sigma_1 p_{j_1+1} \ldots p_m = \min(i_2)\sigma_2 p_{j_2+1} \ldots p_m$.

Assume that $i_1 \geq i_2$. We consider three cases. If $j_1 = j_2$, then we have $\sigma_1 = \sigma_2$ and the equality implies $\min(i_1) = \min(i_2)$ and $i_1 = i_2$. If $j_1 > j_2$, $\min(i_2)\sigma_2$ is a proper prefix of $\min(i_1)$. Let $\delta = |\min(i_1)| - |\min(i_2)\sigma_2|$, then we have $j_2 = j_1 - \delta - 1$. As an occurrence of $\min(i_1)$ ends in $i_1$ (according to lemma 1), an occurrence of $\min(i_2)\sigma_2$ ends in $i_1 - \delta < j_1 - \delta - 1 = j_2$. This is a contradiction with the construction of Oracle($p$). If $j_1 < j_2$, $\min(i_1)$ is a prefix of $\min(i_2)$, so there is an occurrence of $\min(i_1)$ ending before $i_2 \leq i_1$. This is a contradiction with lemma 1.

The three other cases, for $i_1 \leq i_2$, are resolved in a symmetric way. The function is indeed injective, and as the set of proper suffixes of $p = p_1p_2 \ldots p_m$ is of size $m - 1$, $T_{Or}(p) \leq m + m - 1 = 2m - 1$. This maximum is reached for the strings $a^{m-1}b$. □

The factor automaton can be coded in a memory efficient simple way. We do not have to code the states since they are positions in the string. We just have to code the external transitions. Their labels do not have to be coded since they are fixed by their arrival states.

### 2.2 Sequential algorithm

This section presents a sequential construction of the automaton Oracle($p$), that means a way of building the automaton by reading the letters of $p$ one by one from left to right, upgrading the automaton at each step.

We denote $\mathrm{repet}_p(i)$ the longest suffix of $\mathrm{pref}_p(i)$ that appears at least twice in $\mathrm{pref}_p(i)$. We define a function $S_p$ on the states of the automaton, called supply function, that maps each state $i > 0$ of Oracle($p$) to state $j$ in which the reading of $\mathrm{repet}_p(i)$ ends. We arbitrarily set $S_p(0) = -1$. Notice that $S_p(i)$ is well defined for every state $i$ of Oracle($p$) (Corollary 3), and that for any state $i$ of Oracle($p$), $i > S_p(i)$ (lemma 3). We denote $k_0 = m$, $k_i = S_p(k_{i-1})$ for $i \geq 1$. The sequence of the $k_i$ is finite, strictly decreasing and ends in state 0. We denote $CS_p = \{k_0 = m, k_1, \dots, k_t = 0\}$ the suffix path of $p$ in Oracle($p$).

**Lemma 7** *Let $k > 0$ be a state of Oracle(p) such that $s = S_p(k)$ is strictly positive. We denote $w_k = repet_p(k)$ and $w_s = repet_p(s)$. Then $w_s$ is a suffix of $w_k$.*

*Proof.* Let $j$ be the state of Oracle($p$) reached after the reading of $w_s$. Let $v = \min(s)$.

- $j < s$. Lemma 3 implies that $j \leq \text{poccur}(w_s)$ which is by definition strictly less than $s$.
- $|w_s| < |v|$. Assume the contrary. Then there is a path ending by $v = \min(s)$ leading to $j < s$. This is a contradiction with lemma 4.

Lemma 1 implies $s = \text{poccur}(v, p)$. As $w_s$ is a suffix of $\text{pref}_p(s)$ of lenght less than $|v|$, $w_s$ is a proper suffix of $v$, and, as $v$ is himself a suffix of $w_k$ (lemma 2), $w_s$ also is. □

**Corollary 4** *Let $CS_p = \{k_0, k_1 \dots, k_t = 0\}$ be the suffix path of $p$ in Oracle(p) and let $w_i = repet_p(k_{i-1})$ for $1 \leq i \leq t$ and $w_0 = p$. Then, for $0 < l \leq t$, $w_l$ is a suffix of all the $w_i$, $0 \leq i < l \leq t$.*

We now consider for a string $p = p_1 p_2 \dots p_m$ and a letter $\sigma \in \Sigma$ the construction of Oracle($p\sigma$) from Oracle($p$). We denote Oracle($p$) + $\sigma$ the automaton Oracle($p$) on which a transition by $\sigma$ from state $m$ to state $m+1$ is added. We already notice that a transition that exists in Oracle($p$) + $\sigma$ also exists in Oracle($p\sigma$), so that the difference between the two automata only rely on transitions by $\sigma$ to state $m+1$ that have to be added to Oracle($p$) + $\sigma$ in order to get Oracle($p\sigma$). We are investigating states from which there may exist transitions by $\sigma$ to state $m+1$.

**Lemma 8** *Let $k$ be a state of Oracle(p) + $\sigma$ such that there is a transition from $k$ by $\sigma$ to $m+1$ in Oracle($p\sigma$). Then $k$ has to be one of the states of the suffix path $CS_p$ in Oracle(p) + $\sigma$.*

*Proof.* On the contrary, assume that there exists a state $k_{i+1} < k < k_i$ such that there is a transition to $m+1$ by $\sigma$ in Oracle($p\sigma$). We denote $w_j = repet_p(k_{j-1})$ for $1 \leq j \leq t$, and $w_0 = p$. We have $m \in \text{endpos}_p(w_j)$. Let $v = min(k)$. As there is a transition by $\sigma$ from $k$ to $m+1$, $m \in \text{endpos}_p(v)$, and $v$ is comparable to the factors $w_j$ (corollary 4). The factor $v$ must satisfy :

(i) $|v| < |w_i|$. Assume, on the contrary, that $|v| \geq |w_i|$. $|v| > |w_i|$, or else there will be two path labeled by $v$ leading to two different states. Consider the greatest $0 \leq d < i$ such that $|w_{d+1}| < |v| < |w_d|$. The factor $v$ is a suffix of $w_d$. As $v = \min(k)$, according to lemma 1, it occurred in $k < k_d$, and as it is also a suffix of $w_d$, it occurred at least twice in $\mathrm{pref}_p(k_d)$. In that case, by definition of the $k_j$, $k_{d+1}$ can not be $S_p(d)$, and there is a contradiction. So $|v| < |w_i|$ and $v$ is a proper suffix of $w_i$.

(ii) $|v| > |w_{i+1}|$. Assume on the contrary that $|v| \leq |w_{i+1}|$. $|v|$ is then a proper suffix of $w_{i+1}$ and the path labeled $w_{i+1}$ leading to $k_{i+1} < k$ ends by $v = \min(k)$. This is a contradiction with lemma 4.

(iii) $|v| < |\min(k_i)|$. Assume on the contrary that $|v| \geq |\min(k_i)|$. The factor $\min(k_i)$ is a suffix of $w_i$ (lemma 2) of which $v$ is also a suffix (by (i)). So $\min(k_i)$ is a suffix of $v = \min(k)$ and we get a contradiction with corollary 2.

As $v$ is a suffix of $\min(k_i)$ (by (iii)), $k_i \in \mathrm{endpos}_p(v)$ (lemma 1). But $k_i > k \in \mathrm{endpos}_p(v)$ and $v$ occurs at least twice in $\mathrm{pref}_p(k_i)$. As (by (ii))$|v| > |w_{i+1}|$, there is a contradiction with $w_{i+1} = repet_p(k_i)$. □

Among the states on the suffix path of $p$, every state that has no transition by $\sigma$ in Oracle$(p) + \sigma$ must have one in Oracle$(p\sigma)$. More formally, the following lemma sets this fact.

**Lemma 9** *Let $k_l < m$ be a state on the suffix path $CS_p$ of state $m$ in Oracle$(p = p_1p_2 \ldots p_m) + \sigma$. If $k_l$ does not have a transition by $\sigma$ in Oracle$(p)$, then there is a transition by $\sigma$ from $k_l$ to $m+1$ in Oracle$(p\sigma)$.*

*Proof.* Let $v = \min(k_l)$, then $v$ is a suffix of $w_l = repet_p(k_{l-1})$ (lemma 2). As $w_l$ is a suffix of $p$, $w_l\sigma$ is a suffix of $p\sigma$ and $m+1 = \mathrm{poccur}(w_l\sigma)$. According to the construction of Oracle$(p\sigma)$, there is a transition by $\sigma$ from $k_l$ to $m+1$. □

**Lemma 10** *Let $k_l < m$ be a state on the suffix path $CS_p = \{k_0 = m, k_1 \ldots, k_t = 0\}$ of $m$ in Oracle$(p = p_1p_2 \ldots p_m)+\sigma$. If $k_l$ has a transition by $\sigma$ in Oracle$(p)+\sigma$, then all the states $k_i$, $l \leq i \leq t$ also have a transition by $\sigma$ in Oracle$(p) + \sigma$.*

*Proof.* Let $w_l = repet_p(k_{l-1})$. All the $w_i = repet_p(k_{i-1})$, $0 \leq i \leq l$ are suffixes of $w_l$. As $w_l\sigma$ is recognized by Oracle$(p) + \sigma$, by lemma 5, all its suffixes also are. □

The idea of the sequential construction algorithm is the following. According to the three lemmas 8, 9, 10, to transform Oracle$(p) + \sigma$ in Oracle$(p\sigma)$ we only have to go down the suffix path $CS_p = \{k_0 = m, k_1, \ldots, k_t = 0\}$ of state $m$ and while the current state $k_l$ does not have an exiting transition by $\sigma$, a transition by $\sigma$ to $m+1$ should be added (lemma 9). If $k_l$ already has one, the process ends because, according to lemma 10, all the states $k_j$ after $k_l$ on the suffix path already have a transition by $\sigma$.

To add a single letter, the preceding algorithm is enough. But, as we build the automaton by adding the letters of $p$ the one after the other, we must update the supply function $S_{p\sigma}$ of the new automaton Oracle$(p\sigma)$. As (according to the definition of $S_p$), the supply function of states $0 \leq i \leq m$ does not change from Oracle$(p)$ to Oracle$(p\sigma)$, the only thing to do is to compute $S_{p\sigma}(m+1)$. This is done with the following lemma.

**Lemma 11** *If there is a state $k_d$ which is the greatest element of $CS_p = \{k_0 = m, k_1, \ldots, k_t = 0\}$ in Oracle(p) such that there is a transition by $\sigma$ from $k_d$ to a state $s$ in Oracle(p), then $S_{p\sigma}(m+1) = s$ in Oracle($p\sigma$). Else $S_{p\sigma} = 0$.*

*Proof.* Let $w = \mathrm{repet}_{p\sigma}(m+1)$. First assume that there is no such state. As $0 \in CS_P$ in Oracle($p$), there is no transition by $\Sigma$ leaving 0 in Oracle($p$), so $\sigma$ does not occur in $p$ and $w = \epsilon$ and $S_{p\sigma} = 0$.

We now assume that there is a such state $k_d$. Then $w$ is not the empty string and so we can write $w = \alpha\sigma$. Furthermore $k_d < m$ because $m$ is the last state of Oracle($p$). Let $w_j = \mathrm{repet}_p(k_{i-1})$ for $0 < j \leq t$ and $w_0 = p$. We first prove the two following points :

(1) $|\alpha| < |w_{d-1}|$. Conversely assume that $|\alpha| \geq |w_{d-1}|$, the $w_{d-1}$ is a suffix of $\alpha$. As $\alpha\sigma$ is a factor of $p$ (it occurs twice in $p\sigma$), $\alpha\sigma$ is recognized by Oracle($p$) (lemma 5) and $w_{d-1}\sigma$ also is. This contradicts the fact that $d$ is the greatest such that $k_d$ has a transition by $\sigma$ in Oracle($p$).

(2) Let $i$ be the state recognizing $\alpha$ in Oracle($p$); $i$ is strictly inferior to $k_{d-1}$ according to lemma 5 and to the fact that $i$ has a transition by $\sigma$ whereas $k_d$ has not.

We now compare $\alpha$ and $w_d$. The one is a suffix of the other. We get the two following cases :

(1) Assume that $|\alpha| \geq |w_d|$. $i \geq k_d$ because $w_d$ is a suffix of $\alpha$. We conversely prove that $k_d = i$. Assume that $k_d < i$. Then $|w_d| < |\min(i)|$ because otherwise the path labeled by $w_d$ will end by $\min(i)$ and will leads to state strictly before $i$ which will contradicts lemma 4. But $\min(i)$ also occurs in $k_{d-1}$ because : $\min(i)$ and $\min(k_{d-1})$ are comparable (by suffix relation) and on account of corollary 2 $\min(k_{d-1})$ can not be a suffix of $\min(i)$ so $\min(i)$ is a suffix of $\min(k_{d-1})$. So that $\min(i)$ is a suffix of $w_{d-1}$ which occurs twice in $\mathrm{pref}_p(k_{d-1})$ and which is strictly longer than $w_d = \mathrm{repet}_p(k_{d-1})$. This contradicts the definition of $w_d$. So $k_d = i$ with the result that $\alpha\sigma$ leads to the same state as $w_d\sigma : s$.

(2) Now assume that $|\alpha| < |w_d|$ then $i \leq k_d$ because $\alpha$ is a suffix of $w_d$. $|\alpha| \geq |\min(k_d)|$ because as there exists a transition from $k_d$ by $\sigma$ to $s$, $\min(k_d)\sigma$ is both a factor of $p$ and a suffix of $p\sigma$ and $\alpha\sigma$ is the longest of these factors. $\min(k_d)$ is a suffix of $\alpha$ and by lemma 4 $i \geq k_d$. From which $i = k_d$ and $\alpha\sigma$ leads to the same state as $w_d\sigma : s$.

Then the path $\alpha\sigma$ also leads to $s$ and therefore $s = S_{p\sigma}(m+1)$.
□

From these lemmas we can now deduce an algorithm **add_letter** to transform Oracle($p$) in Oracle($p\sigma$). It is given Figure 3.

**Lemma 12** *The algorithm* **add-letter** *really builds Oracle($p\sigma$) from Oracle($p = p_1p_2 \ldots p_m$) and updates the supply function of the new state $m+1$ of Oracle($p\sigma$).*

**Function add_letter**(Oracle($p = p_1 p_2 \dots p_m$), $\sigma$)
1. Create a new state $m + 1$
2. Create a new transition from $m$ to $m + 1$ labeled by $\sigma$
3. $k \leftarrow S_p(m)$
4. **While** $k > -1$ and there is no transition from $k$ by $\sigma$ **Do**
5. Create a new transition from $k$ to $m + 1$ by $\sigma$
6. $k \leftarrow S_p(k)$
7. **End While**
8. **If** $(k = -1)$ **Then** $s \leftarrow 0$
9. **Else** $s \leftarrow$ where leads the transition from $k$ by $\sigma$.
10. $S_{p\sigma}(m + 1) \leftarrow s$
11. **Return** Oracle($p = p_1 p_2 \dots p_m \sigma$)

**Fig. 3.** Add a letter $\sigma$ to Oracle($p = p_1 p_2 \dots p_m$) to get Oracle($p\sigma$)

*Proof.* We go down the suffix path of $p$ in accordance with lemma 9. We stops in accordance with lemma 10 and we update the supply value of state $m + 1$ according to lemma 11. □

The complete algorithm **Oracle-sequential** that builds Oracle($p = p_1 p_2 \dots p_m$) just consists in upgrading the automaton by adding the letters $p_i$ one by one from left to right with the function **add_letter**.

**Theorem 1** *The algorithm* Oracle-sequential*($p = p_1 p_2 \dots p_m$) builds Oracle($p$).*

*Proof.* By induction on string $p$ using lemma 12. □

**Theorem 2** *The complexity of the algorithm* Oracle-sequential*($p = p_1 p_2 \dots p_m$) is $O(m)$ in time and in space*[1]*.*

*Proof.* The algorithm is in $O(m)$ in space. Indeed, all the transitions which are created by the algorithm are transitions of Oracle($p$). Exactly $m + 1$ states are created and a supply value associated to each of these states can be stored in constant space. So the algorithm requires linear space.

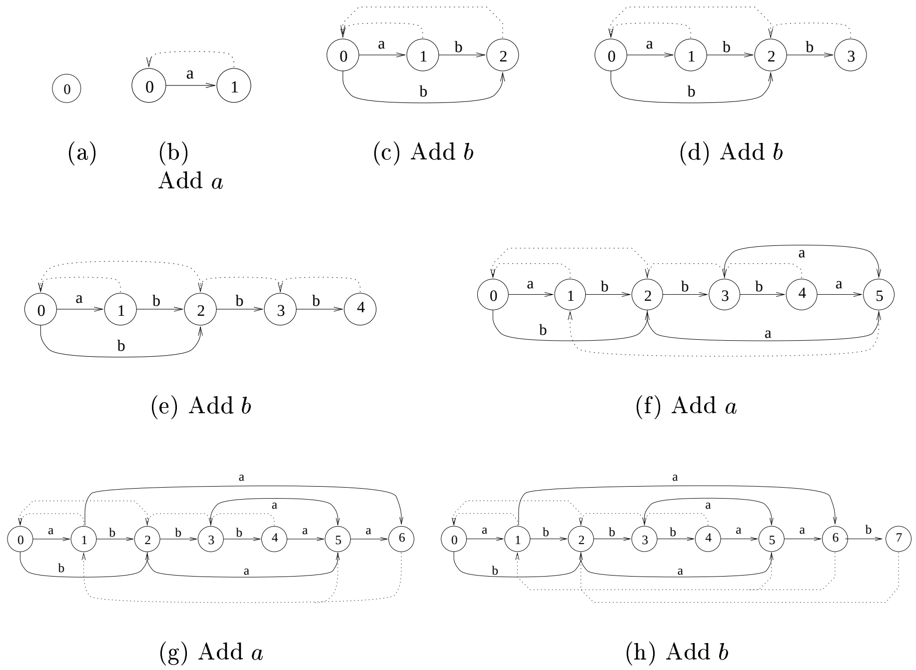
The algorithm is in $O(m)$ in time. As we only create the states and the transitions that are necessary, the only point to verify is that the total number of backward jumps on the supply path (lines 4-6, Figure 3) is linear.

In each stage $i$ of the construction, *i.e.* when letter $p_i$ is being added, the number $r_i$ of backward jumps on the supply path is bounded by $k_i = |\mathrm{repet}_p(i -$

[1] The constants involved in the asymptotic bound of the complexity of the sequential construction algorithm depend on the implementation and may involve the size of the alphabet $\Sigma$. If we implement the transitions in a way that they are accessible in $O(1)$ (use of tables), then the complexity is $O(m)$ in time and $O(|\Sigma| \cdot m)$ in space. If we implement the transitions in a way that they are accessible in $O(log|\Sigma|)$ (use of search trees), then the complexity is $O(log|\Sigma| \cdot m)$ in time and $O(m)$ in space.

1)|. During the transition from stage $i$ to stage $i+1$, we have $k_{i+1} \leq k_i - r_i + 2$ and $r_i \leq k_i - k_{i+1} + 2$. The sum $\sum_{i=1}^{n} r_i$ is therefore bound by $2n$ and the algorithm is linear in time. $\square$

*Example* The sequential construction of Oracle($abbbaab$) is given in Figure 4.



(a) (b) Add $a$ (c) Add $b$ (d) Add $b$

(e) Add $b$ (f) Add $a$

(g) Add $a$ (h) Add $b$

**Fig. 4.** Sequential construction of Oracle(`abbaba`). The dot-lined arrows represent the supply function.

## 3 Suffix oracle

The links between the suffix automaton and our factor oracle lead to a straightforward extension of the oracle : It is possible to mark some states as terminal on the factor oracle as on the suffix automaton in order to recognize suffixes of $p$. This extension will allow us to use some properties of the suffix automaton. We call this new structure *suffix oracle* and we denote it by SOracle($p$).

**Definition 2** *A state $q$ of the suffix oracle is terminal if and only if there is a path labeled by a suffix of $p$ leading from the initial state to $q$.*

The high-level construction algorithm of the factor oracle (see Figure 1) can not be easily modified in order to build the suffix oracle because it can not detect terminal states. Conversely, the sequential construction algorithm can because of the supply function. This is the point of the following lemma. Let us recall that for Oracle($p = p_1 p_2 \dots p_m$), the sequence defined by $k_0 = m$, $k_i = S_p(k_{i-1})$ for $i >= 1$ is finite, strictly decreasing and ends in state 0. We denote $CS_p = \{k_0 = m, k_1 \dots, k_t = 0\}$ the suffix path of $p$ in Oracle($p$).

**Lemma 13** *The terminal states of $SOracle(p)$ are the states of $Oracle(p)$ that are on the suffix path $CS_p$.*

*Proof.*

(i) If $k \in CS_p$, then $k$ is a terminal state of SOracle($p$). We denote $w_0 = p$ and $w_i = repet_p(k_{i-1})$ for all $1 \le i \le t$. Corollary 4 sets that $w_l$ is a suffix of all the $w_i$ with $0 \le i < l \le t$, so $w_l$ is a suffix of $p$. As $w_i$ leads to $k_i$ with $0 \le i \le t$, the states $k_i$ are reachable by a suffix and so are terminal.

(ii) If $q$ is a state of Oracle($p$) such that there is a path from the initial state 0 to $q$ labeled by a suffix $s$ of $p$, then $q \in CS_p$. If $q = m$, $q = k_0$ then the property holds. Assume now that $0 < q < m$. As $s$ is a suffix of $p$, there is an $i$ such that $s$ is a proper suffix of $w_{i-1}$ (which leads to $k_{i-1}$) and such that $w_i$ (which leads to $k_i$) is a proper suffix of $s$. According to lemma 5 we have $k_i \le q \le k_{i-1}$.
Assume that $k_i < q < k_{i-1}$. Consider $v = \min(q)$.
We first deal with the case where $w_i = \epsilon$ ($k_i = 0$). The path labeled by $s$ which leads to $q \neq 0$ ends by $v = \min(q)$ which is not the empty string. As $s$ is a proper suffix of $w_{i-1}$, $v$ is also a suffix of $w_{i-1}$ and $k_{i-1} \in \mathrm{endpos}_p(v)$. So there are occurrences of $v$ both in $q < k_i$ and in $k_i$ which contradicts $\epsilon = w_i = \mathrm{repet}_p(w_{i-1})$.
We can now assume that $|w_i| > 0$.
We first notice that $|v| < |w_i|$. Indeed, conversely assume that $|v| \ge |w_i|$. We can not have $|v| = |w_i|$ because in that case $v = w_i$ (they are both suffixes of $s$) and $q = \mathrm{poccur}(v, p)$. As we assume that $k_i < q$, this is a contradiction with lemma 4. So $|v| > |w_i|$ but in that case as (1) $q \in \mathrm{endpos}_p(v)$ (lemma 1) (2) $k_{i-1} \in \mathrm{endpos}_p(v)$ (lemma 2: $v$ is a suffix of $w_{i-1}$) (3) $q < k_i$ (by assumption), it follows that $v$ is a suffix of $w_{i-1}$, occurs strictly before $k_{i-1}$ and is greater than $w_i$. This contradicts the definition of $w_i = \mathrm{repet}_p(w_{i-1})$.
As $|v| < |w_i|$ and as $w_i$ and $v$ (lemma 2) are suffixes of $s$, $v$ is a proper suffix of $w_i$. As $v = \min(q)$, lemma 4 contradicts the fact that $k_i < q$.

□

To transform the factor oracle of $p$ into a suffix oracle, we just go down the suffix path of the last state created by the sequential construction of Oracle($p$) marking each encountered state as terminal. The pseudo-code of the construction of the suffix oracle (using the sequential construction of the factor oracle) is given in Figure 5.
For instance, the construction of SOracle($abbbaab$) is given in Figure 6.

```
Suffix-oracle(p = p1p2...pm)
    Oracle-sequential(p)
    t ← m
    While Sp(t) ≠ −1 Do
        mark t as terminal
        t ← Sp(t)
    End While
```

**Fig. 5.** Construction algorithm of the suffix oracle SOracle($p = p_1 p_2 \dots p_m$).

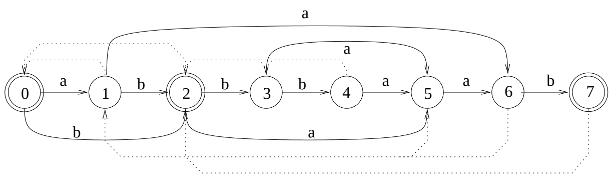



**Fig. 6.** Example of suffix oracle. Double-circled states are terminal. The supply function is represented by dot-lined arrows.

We mainly use the factor oracle rather than the suffix oracle because of the following reason. The power of the structure of factor oracle stands on its simplicity of construction (which becomes a little more complicated for the suffix oracle) but particularly on the few memory which is needed to implement it. This memory saving is based on the fact that the states of the automaton have not to be coded because we can consider a position in the string $p$ as a state. So we only have to code external transitions which are at most $m - 1$. It is more difficult to do the same with the suffix oracle because we need a way of marking the terminal states. This complicates the implementation and slows the terminality test if you want to keep a sharp implementation.

*Note* For some strings, the suffix oracle matches the suffix automaton and therefore recognizes exactly the suffixes. On a binary alphabet $\Sigma = \{0, 1\}$, it is notably the case for Fibonacci words and more generally for any left special factor of an infinite sturmian word (a factor $u$ of a sturmian word is left special if and only if $0u$ and $1u$ are both factors of the same sturmian word). The interested reader can refer to [1] for more details on this point.

## 4 String matching

The factor oracle of $p$ can be used in the same way as the suffix automaton in string matching in order to find the occurrences of a word $p = p_1 p_2 \dots p_m$ in a text $T = t_1 t_2 \dots t_n$ both on an alphabet $\Sigma$. The suffix automaton is used in [9, 8] to get an algorithm called BDM (for *Backward Dawg matching*). Its average complexity is in $O(n \log_{|\Sigma|}(m)/m)$ under a Bernoulli model of probability where all the letters have the same probability. Yao proved in [11] that this bound is optimal. The BDM algorithm moves a window of size $m$ on the text. For each new position of this window, the suffix automaton of $p^r$ (the mirror image of $p$) is used to search for a factor of $p$ from the right to the left of the window. The basic idea of the BDM is that if this backward search failed on a letter $\sigma$ after the reading of a word $u$ then $\sigma u$ is not a factor of $p$ and moving the beginning of the window just after $\sigma$ is secure. This idea is then refined in the BDM using some properties of the suffix automaton.

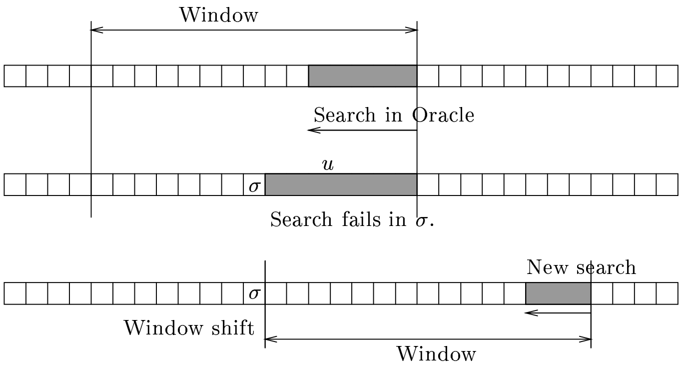



**Fig. 7.** Shift of the search window after the fail of the search by Oracle($p$). The word $\sigma u$ is not a factor of $p$.

However this idea is enough in order to get an efficient string matching algorithm. The most amazing is that the strict recognition of the factors (that the factor and suffix automata allow) is not necessary. For the algorithm to work, it is enough to know that $\sigma u$ is not a factor of $p$. The oracle can be used to replace the suffix automaton as it is illustrated by Figure 7. We call this new algorithm **BOM** for *Backward Oracle Matching*. Its proof is given in lemma 14. We make the conjecture (according to the experimental results) that BOM is still optimal on average.

**Lemma 14** *The BOM algorithm marks all the occurrences of* $p$ *in* $T$ *and only them.*

*Proof.*

– BOM only marks valid occurrences because the only word of length $m$ recognized by Oracle($p^r$) is $p^r$ itself.

```
BOM(p = p1p2...pm, T = t1t2...tn)
  Pre-processing
      Construction of the oracle of p^r
  Search
      pos ← 0
      While (pos <= n − m) do
          state ← initial state of Oracle(p^r)
          j ← m
          While state exists do
              state ← image state by T[pos + j] in Oracle(p^r)
              j ← j − 1
          EndWhile
          If j = 0 do
              mark an occurrence at pos + 1
              j ← 1
          EndIf
          pos ← pos + j
      EndWhile
```

**Fig. 8.** Pseudo-code of **BOM** algorithm.

- BOM marks all the occurrences of $p$. Indeed, assume on the contrary that there is an occurrence of $p$ that no window matches. As the window shift is at most $m$, we have necessarily the situation described in Figure 9 ($u$ may be the empty word) where Window 1 and Window 2 are consecutive in the algorithm. The recognition failure should have occurred in $\sigma$, this is not possible because $\sigma u$ is a factor of $p$.

□

The worst-case complexity of BOM is $O(nm)$. However, in the average, we make the following conjecture based on experimental results (see Section 4.3) :

**Conjecture 1** *Under a model of independence and equiprobability of letters, the BOM algorithm has an optimal average complexity of* $O(n \log_{|\Sigma|}(m)/m)$.

### 4.1 Approach using the suffix oracle

The use of suffix oracle instead of factor oracle allows a refinement of the preceding approach. This refinement comes directly from the use of suffix automaton in BDM. During the backward search phase in suffix oracle, if a terminal state (which does not correspond to the whole word) is encountered, the position in the window is save in a variable *last*. This enables us to give a bound on the longest read factor which is a suffix of $p^r$ ,*i.e.* a prefix of $p$. By saving the last

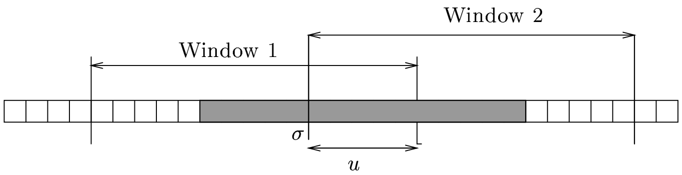



**Fig. 9.** Impossible situation in the BOM algorithm during the search phase.

state, we save a bound on the longest prefix and we can shift the window up to *last* (the string being or not being found in the current window). Figure 10 illustrates this improvement.

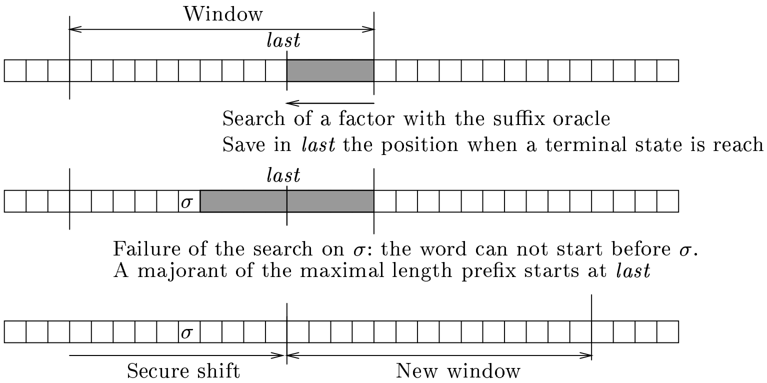



**Fig. 10.** Search with the suffix oracle. The terminal state marking allows an improvement of the search based on the factor oracle.

We call this algorithm **BSOM** for *Backward Suffix Oracle Matching*. Its complexity is still worst case $O(nm)$.

### 4.2 A linear worst case algorithm

Even if the preceding algorithms are very efficient in practice, they have a worst-case complexity in $O(mn)$. There are several techniques to make the BDM algorithm (using suffix automaton) linear in the worst case, and one of them can also be used to make our algorithms linear in the worst case. It uses the Knuth-Morris-Pratt (KMP) algorithm to make a forward reading of some characters in the text.

To explain the combined use of KMP and (factor or suffix) oracle, we consider the current position before the search with the oracle: a prefix $v$ of the string has already been read with KMP at the beginning of the search window and we start the backward search using the oracle from the right end of that current window. The end position of $v$ in the current window is called *critical position* and is denoted by *Critpos*.

The current position is schematized in Figure 11.

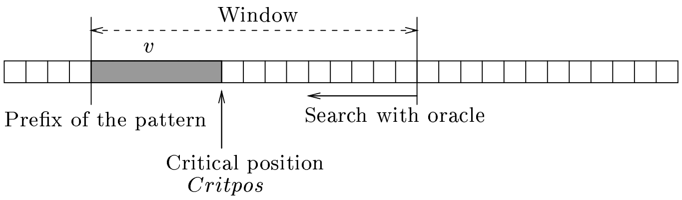



**Fig. 11.** Current position in the linear algorithm using both KMP and (factor or suffix) oracle.

We use the search with the oracle from right to left from the right end of the window. We consider two cases whether the critical position is reached or not.

1. The critical position is not reached. The failure of the recognition of a factor occurs on character $\sigma$ as in the general approach (Figure 12). We shift the window to the left until its beginning goes past character $\sigma$. We restart a KMP search on this new window rereading the characters already read by the oracle. This search stops in a new current position (with a new corresponding critical position) when the recognized prefix is small enough (less than $\alpha m$ with $0 < \alpha < 1$). The value of $\alpha$ is discussed with the experimental results (see section 4.3), typically $\alpha = 1/2$.
   This situation is schematized in Figure 12.

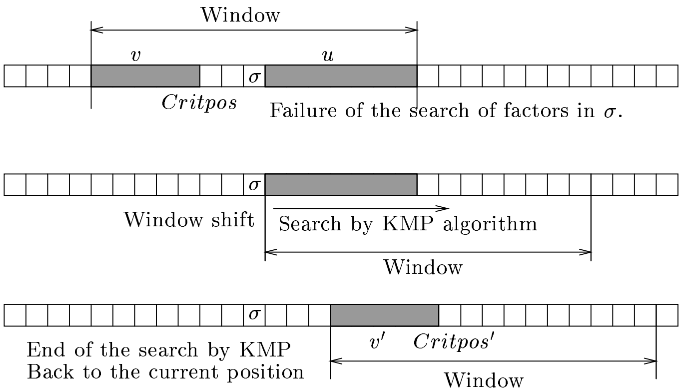



**Fig. 12.** First case : the critical position is not reached.

2. The critical position is reached. We resume the KMP search from the critical position, from the state we were before stopping, rereading at least the characters read by the oracle. We then go on reading the text until the longest recognized prefix is small enough (less than $\alpha m$). This situation is schematized in Figure 13.

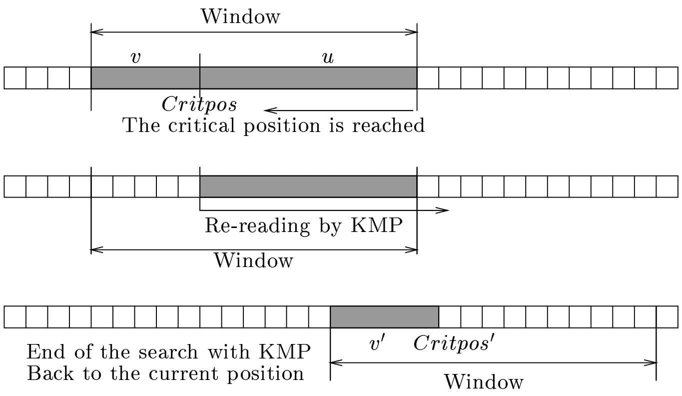



**Fig. 13.** Second case : the critical position is reached

This algorithm can be used with a backward search done with the factor oracle as well as with the suffix oracle (saving the last terminal state encountered). We call these two algorithms **Turbo-BOM** and **Turbo-BSOM**. Concerning the complexity in the worst case, we have the following result.

**Theorem 3** *The two algorithms Turbo-BOM and Turbo-BSOM are: (i) linear considering the number of inspections of characters in the text, the number of these inspections is less than* $2n$*; (ii) linear considering the number of comparisons of characters, the number of these comparisons is less than* $2n$ *when the transitions of the oracle are available in* $O(1)$ *and less than* $2n + n \log |\Sigma|$ *when the transitions are available in* $\log |\Sigma|$.

*Proof.*

(i) Each character in the text is read twice, once during the backward search using the oracle and once during the search with the KMP algorithm. The results follows.

(ii) This complexity comes directly from the fact that the number of comparisons done by the KMP algorithm is less than $2n$. If the transitions of the oracle are available in $O(1)$, the backward search using the oracle does not require any comparisons (it only requires inspections), and the total number of comparisons is bound by the number of comparisons in KMP : $2n$. If the transitions are available in $\log \Sigma$ (for instance with binary search trees), the number of comparisons done during the backward search is bounded by $n \log \Sigma$ and the total number by $2n + n \log \Sigma$.

□

### 4.3 Experimental results

In this section, we present experimental results on the time complexity of our string matching algorithms, compared to the following algorithms: **Sunday**: the Sunday algorithm [10] is often considered as the fastest in practice; **BM**: the Boyer-Moore algorithm [6]; **BDM**: the classical Backward Dawg Matching with a suffix automaton [8]; **Suff**: the Backward Dawg Matching with a suffix automaton but without testing terminal states, this is equivalent to the basic approach with the factor automaton[2]; **BOM**: the Backward Oracle Matching with the factor oracle; **BSOM**: the Backward Oracle Matching with the suffix oracle (testing terminal states); **Turbo-BOM**: the linear algorithm using BOM and KMP with $\alpha = 1/2$.

Our string matching experiments are done on DNA sequences (we took the Archaeoglobus Fulgidus sequence of 2 Mb) and on natural language (English - we took a compilation of Wall Street Journal articles of 10 Mb). Results are obtain with an accuracy of +/- 2% with a confidence of 95% (which may require thousands of iterations). We also performed experiments on random texts, that are not shown in this version. The machine used is a PC with a Pentium II processor at 350MHz running Linux 2.0.32 operating system. For all the algorithms, the transitions of the automata are implemented as tables which allow $O(1)$ branches.

Experimental results in string matching are always surprising because codes are smalls and the time taken by a character comparison is not much greater than the time taken by an integer incrementation. It is for instance the reason why Sunday algorithm is the fastest algorithm for small strings: a window shift is usually very small but require very few operations. It is also the reason why BDM is slower than Suff and BSOM slower than BOM whereas the window shifts in BSOM and BDM are greater. When searching in sequences of characters, it is obviously useless to mark and test terminal states in both suffix automaton and factor oracle.

The 2 sub-figures of Figure 15 show that BOM is as fast as Suff (except on a binary alphabet) which is much more complicated and requires much more

[2] The suffix automaton without taking in account the terminal states (i.e. considering every state as terminal) and the factor automaton recognize the same language. The difference is that the factor automaton is minimal, so its size is smaller or equal than the size of the suffix automaton. But the difference of size is not significant in practice, anyway not enough significant to justify the implementation of a factor automaton which will complicate and slow the preprocessing phase of the string matching algorithm.

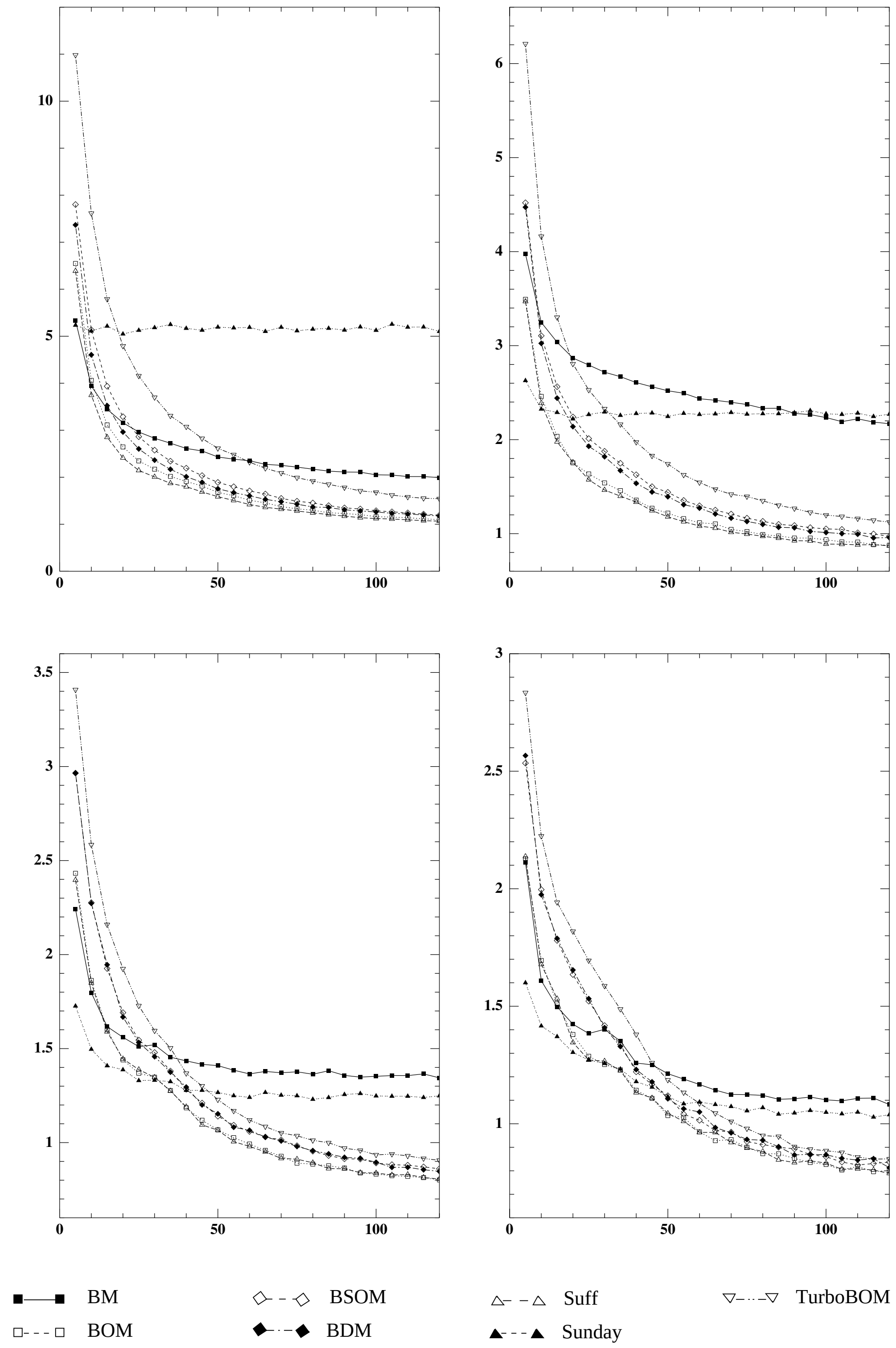



**Fig. 14.** Experimental results in time of the string matching algorithms on random texts of size 10 Mb on alphabets of size 2, 4, 16 and 32. The X-axis represents the length of the string and the Y-axis the search time in 1/100th seconds per Mbytes

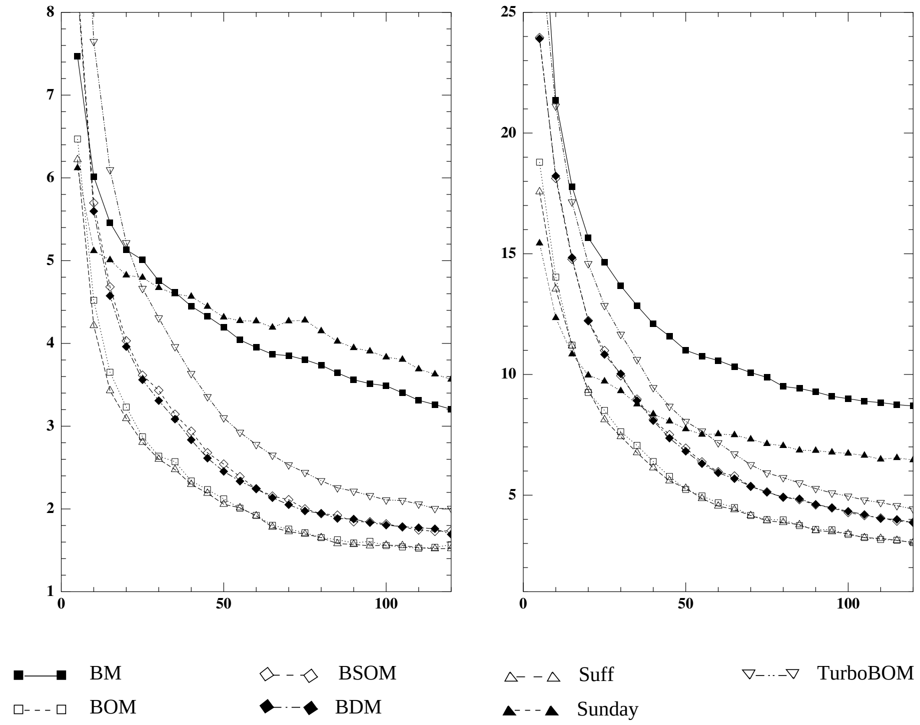



**Fig. 15.** Experimental results in time of the string matching algorithms on DNA sequences (left plot - 2Mb) and Natural language (right plot - 10 Mb). The X-axis represents the length of the string and the Y-axis the search time in 1/100th seconds per Mbytes.

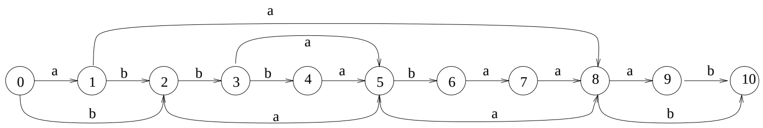
memory. BOM reads more texts characters than Suff, but as the oracle automaton is much smaller than the suffix automaton, it performs less memory page breaks and the experimental search times are the sames.

Turbo-BOM algorithm is the slowest but it is the only one that can be used in real time and in that case its behavior is rather good. It has to be noticed that we arbitrarily set the value of $\alpha$ to $1/2$. However, according to the tests we performed for different values of $\alpha$, it turns out that $\alpha = 1/2$ is the more often the best value and that the variations of search times with other values of $\alpha$ (as far as they stay between $(2\log_{|\Sigma|} m)/m$ and $(m - 2\log_{|\Sigma|} m)/m$ ) are not very significant and anyway do not deserve by themselves an accurate study.

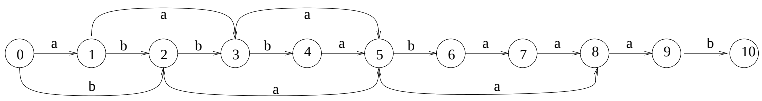
## 5 Conclusions

The two new structures we presented, the factor oracle and the suffix oracle, enable new string matching algorithms. These algorithms are very efficient in practice, as efficient as the ones which already existed, but are far more simple to implement and require less memory. According to the experimental results, we conjecture that they are optimal on the average (under a model of equiprobability of letters) but it remains to be shown.

About the structure of factor oracle itself, many questions stay open. Among others, it would be interesting to have a characterization of the language recognized by the oracle. It would also be of interest to study of the average number of external transitions in the oracle, to know the average memory space required by the string matching algorithms.




(a) Factor oracle




(b) Reduced automaton

**Fig. 16.** The factor oracle is not minimal considering the number of transitions among the automata of $m+1$ states which recognize at least the factors.

We notice that the factor oracle is not minimal considering the number of transitions among the automata of $m+1$ states which recognize at least the factors. This reduced automaton may also be used in string matching provided that its construction can be done in linear time. This construction remains an open problem.

Finally, the factor oracle can be extended to a set of strings, and integrated in multi string matching algorithms. The experimental results are very promising, the new algorithms being by far the fastest in many practical cases [3].

## References

1. C. Allauzen. *Combinatoire sur les mots et recherche de motifs (Combinatorics on words and string matching).* PhD thesis, Université de Marne-la-Vallée, 2001.
2. C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle: a new structure for pattern matching. In Miroslav Bartosek Jan Pavelka, Gerard Tel, editor, *SOFSEM'99, Theory and Practice of Informatics (Brno, 1999)*, number 1725 in LNCS, pages 291–306. Springer-Verlag, 1999.

3. C. Allauzen and M. Raffinot. Factor oracle of a set of words. Technical Report 99-11, Institut Gaspard-Monge, Université de Marne-la-Vallée, 1999. http://www-igm.univ-mlv.fr/~raffinot/ftp/IGM99-11-english.ps.gz.
4. R. A. Baeza-Yates. Searching subsequences. *Theor. Comput. Sci.*, 78(2):363–376, 1991.
5. A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M. T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40(1):31–55, 1985.
6. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
7. M. Crochemore. Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86, 1986.
8. M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.
9. A. Czumaj, M. Crochemore, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12:247–267, 1994.
10. D. Sunday. A very fast substring search algorithm. *CACM*, 33(8):132–142, August 1990.
11. A. C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.