Factor oracle: a new structure for pattern matching

Cyril Allauzen Maxime Crochemore* Mathieu Raffinot

Institut Gaspard-Monge, Université de Marne-la-Vallée,
77454 Marne-la-Vallée Cedex 2, France.
http://www-igm.univ-mlv.fr/LabInfo/
{allauzen,mac,raffinot}@monge.univ-mlv.fr

Abstract We introduce a new automaton on a word p, sequence of letters taken in an alphabet Σ, that we call *factor oracle*. This automaton is acyclic, recognizes at least the factors of p, has $m+1$ states and a linear number of transitions. We give an on-line construction to build it. We use this new structure in string matching algorithms that we conjecture optimal according to the experimental results. These algorithms are as efficient as the ones that already exist using less memory and being more easy to implement.

Keywords: indexing, finite automaton, pattern matching, algorithm design.

1 Introduction

A *word* p is a finite sequence $p = p_1 p_2 \ldots p_m$ of letters taken in an alphabet Σ. We keep the notation p along this paper to denote the word on which we are working.

Efficient pattern matching on fixed texts are based on indexes built on top of the text. Many indexing techniques exist for this purpose. The simplest methods use precomputed tables of q-grams while more achieved methods use more elaborated data structures. These classical structures are: suffix arrays, suffix trees, suffix automata or DAWGs[1], and factor automata (see [11]). When regarded as automata, they accept the set of factors (substrings) of the text. All these structures lead to very time-efficient pattern matching algorithms but require a fairly large amount of memory space. It is considered, for example, that the implementation of suffix arrays can be achieved using five bytes per text character and that other structures need about twelve bytes per text character.

Several strategies have been developed to reduce the memory space required to implement structures for indexes.

* Work by this author is supported in part by Programme "Génomes" of C.N.R.S.

[1] DAWGs, Directed Acyclic Word Graphs, are just suffix automata in which all states are terminal states

One of the oldest method is to merge the compression techniques applied both by the suffix tree and the suffix automaton. It leads to the notion of compact suffix automaton (or compact DAWG) [5]. The direct construction of this structure is given in [12, 13].

A second method to reduce the size of indexes has been considered in the text compression method in [10]. It consists in representing the complement language of the factors (substrings) of the text. More precisely, only minimal factors not occurring in the text need to be considered [9, 8]. Which allow to store them in a tree and to save space.

We present in this paper a third method. We want to build an automaton (a) that is acyclic (b) that recognizes at least the factors of p (c) that has the fewer states as possible and (d) that has a linear number of transitions. We already notice that such an automaton has necessarily at least $m + 1$ states.

The suffix or factor automaton [4, 7] satifies (a)-(b)-(d) but not (c) whereas the sub-sequence automaton [3] satisfies (a)-(b)-(c) but not (d), which makes the problem non trivial.

We propose an intermediate structure that we call the *factor oracle* : an automaton with $m + 1$ states that satisfies these four requirements.

We use this new structure to design new string matching algorithms. These algorithms have a very good average behaviour that we conjecture as optimal. The main advantages of these new algorithms are (1) that they are easy to implement for an optimal behaviour and (2) the memory saving that the factor oracle allows with respect to the suffix automaton. The structure has been extended in [2] to implement the index of a finite set of texts.

The paper is structured as follows: Section 2 discusses the construction of the factor oracle, Section 3 describes a string matching based on the factor oracle and shows experimental results, and finally we conclude in Section 4. Proofs of the results presented in the paper may be found in [1]. We now define notions and definitions that we need along this paper.

A word $x \in \Sigma^*$ is a *factor* of p if and only if p can be written $p = uxv$ with $u, v \in \Sigma^*$. We denote $\mathrm{Fact}(p)$ the set of all the factors of word 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 word ϵ.

We denote $\mathrm{pref}_p(i)$ the prefix of length i of p for $0 \le i \le |p|$.

We denote for $u \in \mathrm{Fact}(p)$, $\mathrm{poccur}(u, p) = \min\{|z|\ ,\ z = wu \text{ et } p = wuv\}$, the ending position of the first occurrence of u in p.

Finally, we define for $u \in \mathrm{Fact}(p)$ the set $\mathrm{endpos}_p(u) = \{i \mid p = wup_{i+1} \dots p_m\}$. If two factors u and v of p are such that $\mathrm{endpos}_p(u) = \mathrm{endpos}_p(v)$, we denote $u \sim_p v$. It is very easy to verify that $\sim_p$ is an equivalence relation; it is in fact the syntaxic equivalence of the language $\mathrm{Suff}(p)$.

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 σ

Figure1. High-level construction algorithm of the Oracle

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

The factor oracle of the word $p = abbbaab$ is given as an example figure 2. On this example, it can be noticed that the word aba is recognized whereas it is not a factor of p.

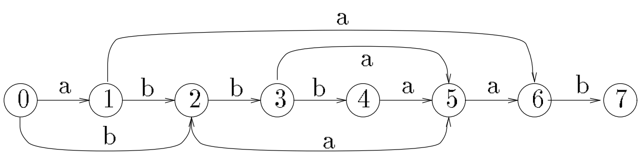


Figure2. Factor oracle of $abbbaab$. The word aba is recognizes whereas it is not a factor.

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 word among the words recognized in state* i *of* $Oracle(p)$*. Then,* $u \in Fact(p)$ *and* $i = poccur(u, p)$.

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

We denote $\min(i)$ the minimal length word of 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.*

Lemma 2 *Let i be a state of Oracle(p) and $u = min(i)$. u is a suffix of any word $c \in \Sigma^*$ which is the label of a path leading from state 0 to state i.*

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

Note: In lemma 3, j is really less or equal than poccur(w, p), and not always equal. The example given figure 3 represents the automaton Oracle($abbcabc$), and the state reached after the reading of the word abc is strictly less than poccur$(abc, abbcabc)$.

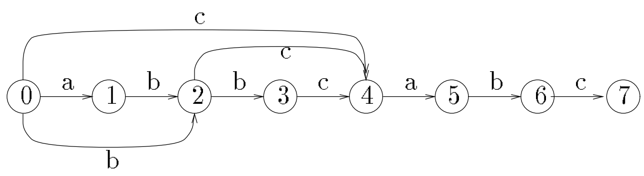


Figure3. Example of a factor (abc) that is not recognized at the end of his first occurrence but before.

Corollary 3 *Let $w \in Fact(p)$. Every word $v \in Suff(w)$ 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 $j \geq i$.*

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

The number of states of Oracle(p) with $p = p_1 p_2 \ldots p_m$ is $m + 1$. We now consider the number of transitions.

Lemma 6 *The number $T_{Or}(p)$ of transitions in Oracle($p = p_1 p_2 \ldots p_m$) satisfies $m \leq T_{Or}(p) \leq 2m - 1$.*

2.2 On-line algorithm

This section presents an on-line 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.

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 defined 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 $S_p(0) = -1$.

Notes:

- $S_p(i)$ is well defined for every state i of Oracle(p) (Corollary 3).
- 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

$$CS_p = \{k_0 = m, k_1, \ldots, k_t = 0\}$$

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.*

Corollary 4 *Let $CS_p = \{k_0, k_1 \ldots, k_t = 0\}$ 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 word $p = p_1 p_2 \ldots p_m$ 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 σ 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 may only rely on transitions by σ 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 be transitions by σ to state $m+1$.

Lemma 8 *Let k be a state of Oracle(p) $+ \sigma$ such that there is a transition from k by σ to $m+1$ in Oracle($p\sigma$). Then k has to be one of the states on the suffix path $CS_p = \{k_0 = m, k_1, \ldots, k_t = 0\}$ in Oracle(p) $+ \sigma$.*

Among the states on the suffix path of p, every state that has no transition by σ 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 $CS_p = \{k_0 = m, k_1, \ldots, k_t = 0\}$ of state m in Oracle($p = p_1 p_2 \ldots p_m$) $+ \sigma$. If k_l does not have a transition by σ in Oracle(p), then there is a transition by σ from k_l to $m+1$ in Oracle($p\sigma$).*

Lemma 10 *Let $k_l < m$ be a state on the suffix path $CS_p = \{k_0 = m, k_1, \ldots, k_t = 0\}$ in Oracle($p = p_1 p_2 \ldots p_m$) $+ \sigma$. If k_l has a transition by σ in Oracle(p) $+ \sigma$, then all the states k_i, $0 \leq i \leq t$ also have a transition by σ in Oracle(p) $+ \sigma$.*

The idea of the on-line 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 σ, a transition by σ 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 σ.

If we only wanted to add a single letter, the preceding algorithm would be enough. But, as we want to be able to build the automaton by adding the letters of p the one after the other, we have to be able to 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 σ from k_d to a state s in Oracle(p), then $S_{p\sigma}(m+1) = s$ in Oracle($p\sigma$). Else $S_{p\sigma} = 0$.*

From these lemmas we can now deduce an algorithm **add_letter** to transform Oracle(p) in Oracle($p\sigma$). It is given figure 4.

Lemma 12 *The algorithm* **add-letter** *really builds Oracle($p = p_1 p_2 \ldots p_m \sigma$) from Oracle($p = p_1 p_2 \ldots p_m$) and update the supply function of the new state $m+1$ of Oracle($p\sigma$).*

The complete on-line algorithm to build Oracle($p = p_1 p_2 \ldots p_m$) just consits in adding the letters p_i one by one from left to right. It is given figure 5.

Theorem 1 *The algorithm* Oracle-on-line*($p = p_1 p_2 \ldots p_m$) builds Oracle(p).*

Theorem 2 *The complexity of* Oracle-on-line*($p = p_1 p_2 \ldots p_m$) is $O(m)$ in time and in space.*

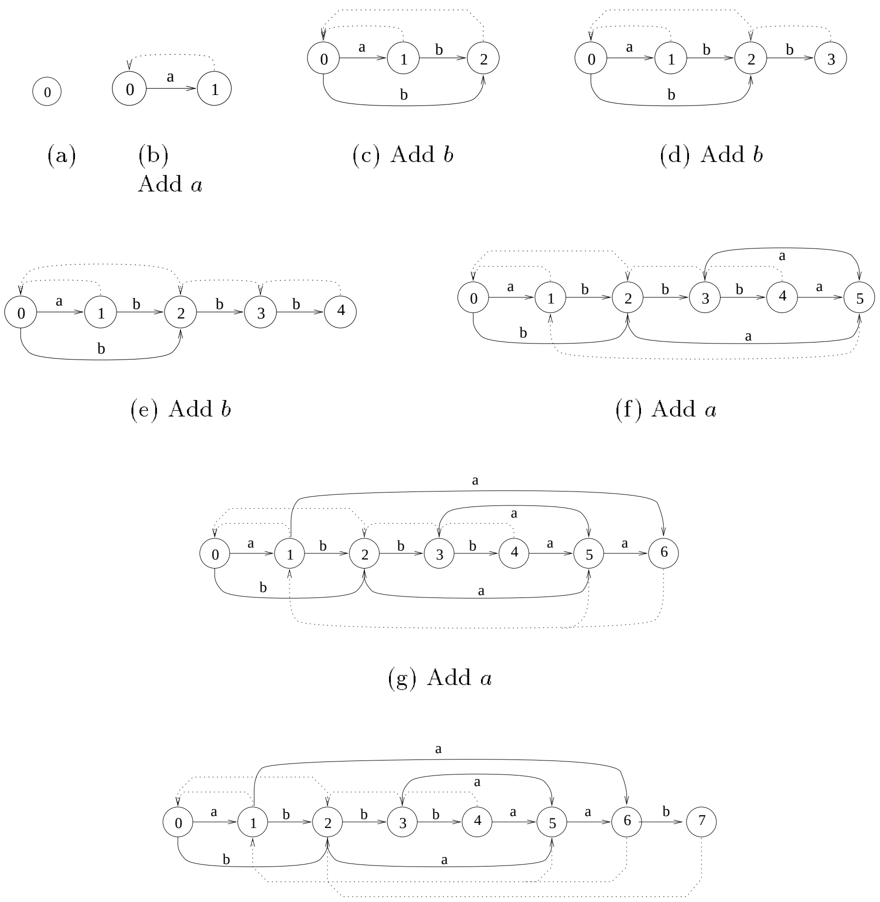
Note The constants involved in the asymptotic bound of the complexity of the on-line construction algorithm depend on the implementation and may involve the size of the alphabet Σ. If we implement the transitions in a way that they are accessible in $O(1)$ (use of tables), then the complexity is $O(m)$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

Fonction add_letter(Oracle($p = p_1p_2\ldots p_m$), σ)
Create a new state $m+1$
Create a new transition from m to $m+1$ labeled by σ
$k \leftarrow S_p(m)$
While $k > -1$ and there is no transition from k by σ **Do**
        Create a new transition from k to $m+1$ by σ
        $k \leftarrow S_p(k)$
End While
If $(k = -1)$ **Then** $s \leftarrow 0$
Else $s \leftarrow$ where leads the transition from k by σ.
$S_{p\sigma}(m+1) \leftarrow s$
Return Oracle($p = p_1p_2\ldots p_m\sigma$)

Figure4. Add a letter σ to Oracle($p = p_1p_2\ldots p_m$) to get Oracle($p\sigma$)

Oracle-on-line($p = p_1p_2\ldots p_m$)
Create Oracle(ϵ) with:
        one single state 0
        $S_\epsilon(0) \leftarrow -1$
For $i \leftarrow 1$ à m **Do**
    Oracle($p = p_1p_2\ldots p_i$) $\leftarrow$ `add_letter`(Oracle($p = p_1p_2\ldots p_{i-1}$),p_i)
End For

Figure5. On-line construction algorithm of Oracle($p = p_1p_2\ldots p_m$).



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

(e) Add b (f) Add a

(g) Add a

(h) Add b

Figure6. On-line construction of Oracle(`abbaba`). The dot-lined arrows represent the supply function.

Exemple The on-line construction of Oracle($abbbaab$) is given figure 6.

3 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 $p = p_1 p_2 \ldots p_m$ in a text $T = t_1 t_2 \ldots t_n$ both on an alphabet Σ.

The suffix automaton is used in [14, 11] to get an optimal algorithm in the average called BDM (for *Backward Dawg matching*). Its average complexity is in $O(n \log_{|\Sigma|}(m)/m)$ under a Bernouilli model of probability where all the letters are equiprobable.

The BDM algorithm move 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 σ after the reading of a word u then σu is not a factor of p and moving the beginning of the window just after σ is secure. This idea is then refined in the BDM using some properties of the suffix automaton.

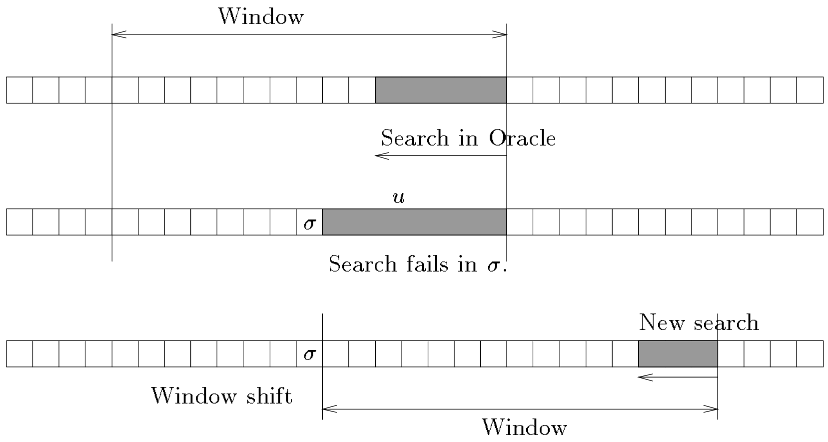


Figure7. Shift of the search window after the fail of the search by Oracle(p). The word σu is not a factor of p.

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 $u\sigma$ 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*. The pseudo-code of BOM is given figure 3. Its proof is given lemma 13. We make the conjecture (according to the experimental results) that BOM is still optimal in the average.

```
BOM(p = p_1 p_2 ... p_m, T = t_1 t_2 ... t_n)
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
```

Figure8. Pseudo-code of **BOM** algorithm.

Lemma 13 *The BOM algorithm marks all the occurrences of p in T and only them.*

The worst-case complexity of BOM is $O(nm)$. However, in the average, we make the following conjecture based on experimental results (see 3.2) :

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

3.1 A linear algorithm in the worst case

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 pattern has already be 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 at figure 9.

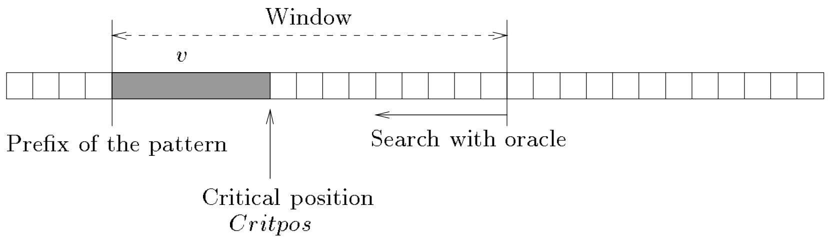


Figure9. Current position in the linear algorithm using both KMP and (factor or suffix) oracle.

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 σ as in the general approach (figure 7). We shift the window to the left until its beginning goes past character σ. 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 αm with $0 < \alpha < 1$). The value of α is discussed with the experimental results (see section 3.2), typically $\alpha = 1/2$. This situation is schematized figure 10.
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 α). This situation is schematized figure 11.

This algorithm can be used with a backward search done with the factor oracle. We call this new algorithm **Turbo-BOM**. Concerning the complexity in the worst case, we have the following result.

Theorem 3 *The algorithm Turbo-BOM is*

(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* $\log\Sigma$.

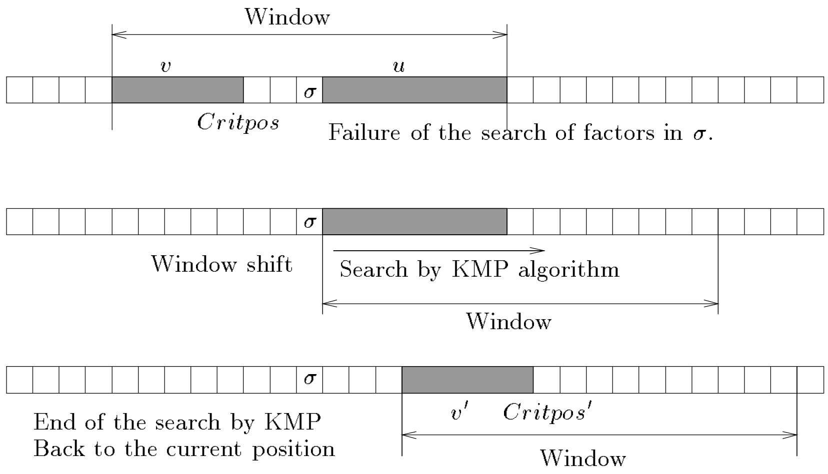


Figure10. First case : the critical position is not reached.

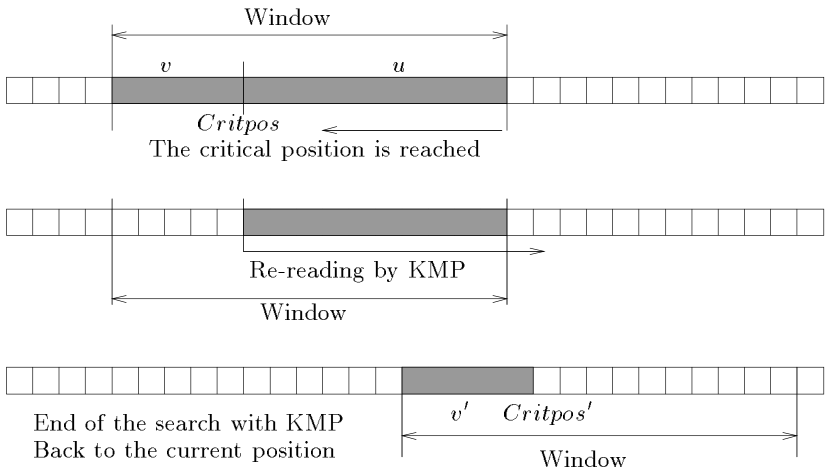


Figure11. Second case : the critical position is reached

3.2 Experimental results

In this section, we present the experimental results obtained. More precisely, we compare the following algorithms.

- **Sunday**: the Sunday algorithm [15] is often considered as the fastest in practice,
- **BM**: the Boyer-Moore algorithm [6],
- **BDM**: the classical Backward Dawg Matching with a suffix automaton [11],
- **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. This later structure is not described in this version of the paper, but can be found in [1].
- **Turbo-BOM**: the linear algorithm using BOM and KMP with $\alpha = 1/2$.

Our string matching experiments are done on random texts of size 10 Mb with an accuracy of +/- 2% with a confidence of 95% (which may require thousands of iterations) for alphabets of size 2, 4, 16 and 32. 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 $O(1)$ branchs. But it is not realistic (especially for the suffix automaton) when the alphabet becomes rather big (for instance for 16 bits character coding). Moreover, the Sunday algorithm becomes unusable as it is when the alphabet is big because it mainly uses character table.

Experimental results in string matching are always surprising because codes are small and the time taken by a comparison is not much greater than the time taken by an indice incrementation. It is for instance the reason why Sunday algorithm (when it is usable) is the fastest algorithm for small patterns. The window shift are very small but very few operations are necessary to get this shift. It is also the reason why BDM is slower than Suff whereas the window shifts in BSOM and BDM are greater.

The 4 subfigures of figure 12 shows that BOM is as fast as Suff (except on a binary alphabet) which is much more complicated and requires much more memory.

It is obviously useless (in the case of searchs in texts of characters) to mark and test terminal states in both suffix automaton and factor oracle.

[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.

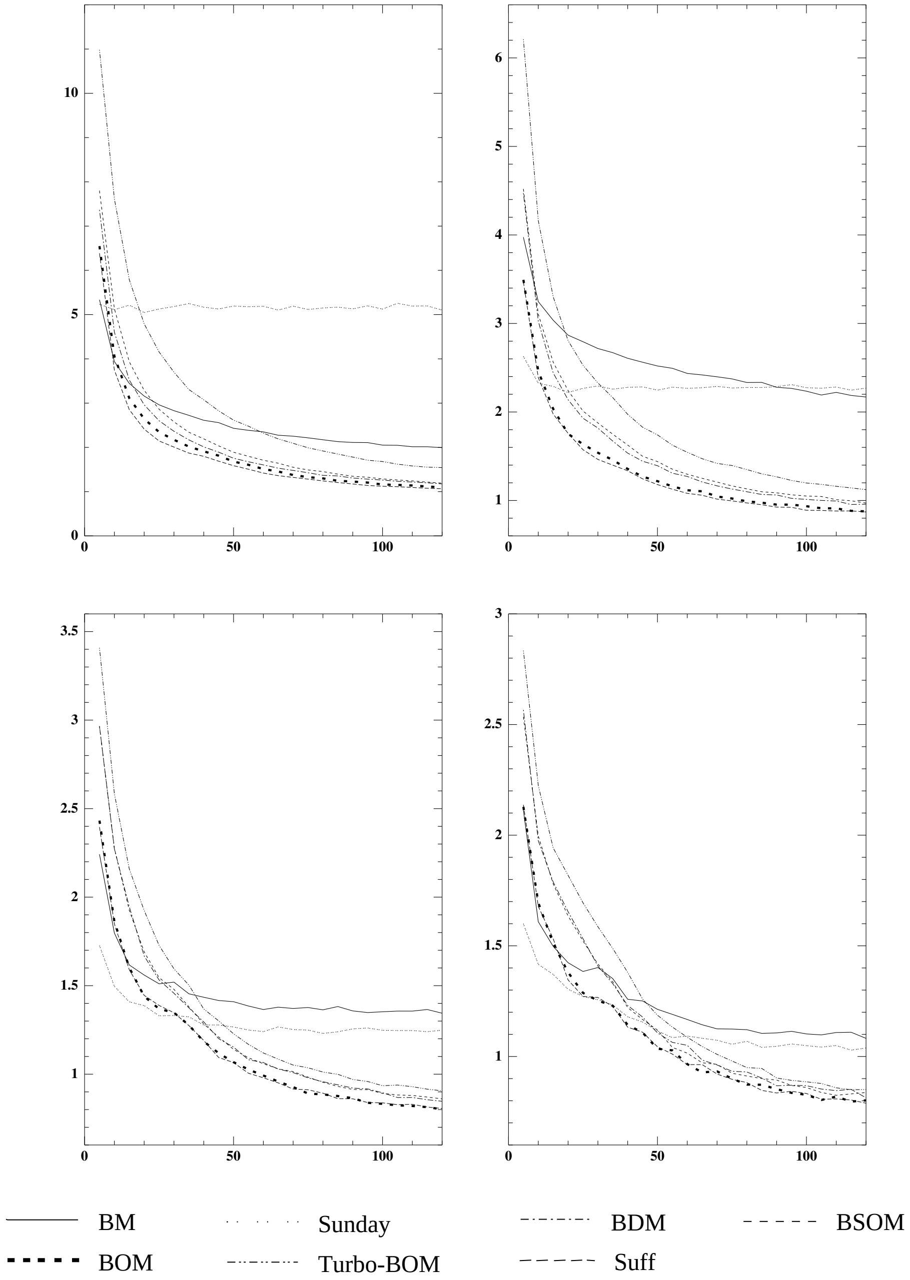


Figure12. Experimental results in time of the string matching algorithms on random texts of size 10 Mb on alphabets of size 2, 4, 16 and 32. The X-axis represents the length of the pattern and the Y-axis the search time in 1/100th seconds per Mbytes

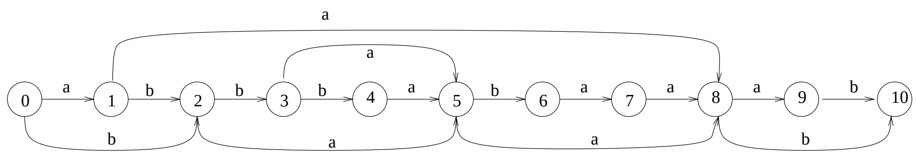
Turbo-BOM algorithm is the slowest but it is the only one that can be used in real time and in that case its behavior is rather good. It has to be noticed that we arbitrarily set the value of α to $1/2$. However, according to the tests we have proceeded for different values of α, it turns out that $\alpha = 1/2$ is the more often the best value and that the variations of search times with other values of α (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.

4 Conclusions

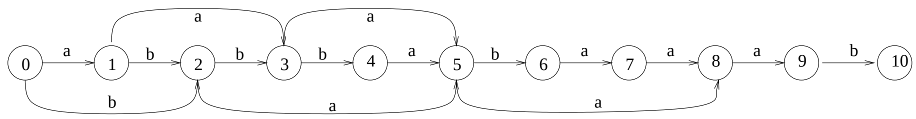
The new structure we presented, the *factor oracle*, allow new string matching algorithms. These algorithms are very efficient in practice, as efficient as the ones which already exists, but are far more simple to implement and require less memory. According to the experimental results, we conjecture 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 interesting to have a study of the average number of external transitions in the oracle. It would give an idea of the average memory space required by the string matching algorithms.



(a) Factor oracle



(b) Reduced automaton

Figure13. The factor oracle is not minimal considering the number of transitions among the automata of $m + 1$ states which recognize at least the factors.

Finally, 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. An example is given figure 13. 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.

References

1. C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle, Suffix oracle. Technical Report 99-08, Institut Gaspard-Monge, Université de Marne-la-Vallée, 1999. http://www-igm.univ-mlv.fr/~raffinot/ftp/IGM99-08-english.ps.gz.
2. C. Allauzen and M. Raffinot. Oracle des facteurs d'un ensemble de mots. Rapport technique 99-11, Institut Gaspard Monge, Université de Marne-la-Vallée, 1999. http://www-igm.univ-mlv.fr/~raffinot/ftp/IGM99-11.ps.gz.
3. R. A. Baeza-Yates. Searching subsequences. *Theor. Comput. Sci.*, 78(2):363–376, 1991.
4. 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.
5. A. Blumer, A. Ehrenfeucht, and D. Haussler. Average size of suffix trees and DAWGS. *Discret. Appl. Math.*, 24:37–45, 1989.
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, F. Mignosi, and A. Restivo. Automata and forbidden words. *Information Processing Letters*, 67(3):111–117, 1998.
9. M. Crochemore, F. Mignosi, and A. Restivo. Minimal forbidden words and factor automata. In L. Brim, J. Gruska, and J. Zlatuška, editors, *Mathematical Foundations of Computer Science 1998*, number 1450 in LNCS, pages 665–673. Springer-Verlag, 1998. extended abstract of [8].
10. M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Text compression using antidictonaries. Rapport I.G.M. 98-10, Université de Marne-la-Vallée, 1998.
11. M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.
12. M. Crochemore and R. Vérin. Direct construction of compact directed acyclic word graphs. In A Apostolico and J. Hein, editors, *Combinatorial Pattern Matching*, number 1264 in LNCS, pages 116–129. Springer-Verlag, 1997.
13. M. Crochemore and R. Vérin. On compact directed acyclic word graphs. In J. Mycielski, G. Rozenberg, and A. Salomaa, editors, *Structures in Logic and Computer Science*, number 1261 in LNCS, pages 192–211. Springer-Verlag, 1997.
14. 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.
15. D. Sunday. A very fast substring search algorithm. *CACM*, 33(8):132–142, August 1990.