1

Different applications of the factor oracle: an overview

T. Lecroq and A. Lefebvre

ABSTRACT. We present in this paper different applications of the factor oracle of a string: string-matching, detection of repeats in large genomic sequences and linear on-line lossless data compression.

1 Introduction

The factor oracle of a string x of length m, introduced in [1, 2] is an acyclic automaton which recognizes at least all the factors of x and some more strings. Actually its language contains the set of all the factors of x and is contained by the set of all the subsequences of x. Its exact characterization still remains an open problem.

This data structure has the particularity to be defined from its construction algorithm. Its implementation is very easy and its construction time and space are very economical since it has exactly $m+1$ states and between m and $2m-1$ transitions.

The purpose of this article is to present an overview of the construction methods of the factor oracle and of its different applications.

The following of this article is organized as follows. First we recall some notations and basic definitions. In section 3 we present different constructions of the factor oracle: the first two constructions are off-line while the last one is on-line. In section 4 we introduce the suffix oracle of a string. Section 5 is devoted to string matching: four algorithms are presented that use either the factor oracle or the suffix oracle. Section 6 deals with the detection of repeats in genomic sequences. In section 7 it is shown how the detection of repeats can be used to perform linear on-line lossless data compression. Finally section 8 gives the conclusion and lists some open problems.

2 Notations

A *string* is a sequence of zero or more symbols from an *alphabet* Σ; the elements of Σ are called *characters*, *letters* or *symbols*. the string with zero symbols is denoted by ε. The set of all strings over the alphabet Σ is denoted by Σ^*. A string x of length m is represented by $x[1..m]$, where

NATO book, 1–26.

$x[i] \in \Sigma$ for $1 \leq i \leq m$. A position on a string x of length m is an integer such as $1 \leq i \leq m$. The reverse of a string x of length m is denoted by x^R and is equal to the sequence of characters of x read from right to left $x[m]x[m-1]\cdots x[1]$. A string w is a *factor* of x if $x = uwv$ for $u, v \in \Sigma^*$; we equivalently say that the string w occurs at position $|u|+1$ of the string x. The position $|u|+1$ is said to be the *starting position* of w in x and the position $|u|+|w|$ the *ending position* of w in x. The string $x[i..j]$ denotes the factor of x with the starting position i and the ending position j whenever $i \leq j$ and $x[i..j] = \varepsilon$ when $j < i$. A string w is a *prefix* of x if $x = wu$ for $u \in \Sigma^*$. Similarly, w is a *suffix* of x if $x = uw$ for $u \in \Sigma^*$. A string w is a *border* of x if w is both a prefix and a suffix of x: $x = uw = wv$ for $u, v \in \Sigma^*$. In this case the integer $|u| = |v|$ is said to be a *period* of x. *The border* of x is the longer among all its proper borders. *The period* of x is the smallest among its non-null periods.

The set of all the factors of x is denoted by $Fact(x)$.

We say that a string w is a *subsequence* of x if there exist $|w|+1$ strings $u_1, u_2, \ldots, u_{|w|+1}$ such that $x = u_1 w[1] u_2 w[2] \ldots w[|w|] u_{|w|+1}$; in words, w can be obtained from x by deleting $|x| - |w|$ characters.

3 Different constructions of the factor oracle

In this section we present different methods for constructing the factor oracle of a string x of length m. Along the line we introduce several properties of this oracle.

3.1 Off-line constructions

A first construction

We first start with an off-line method for constructing the factor oracle of x introduced in [1]. Actually the oracle is defined from this construction.

We introduce a notation for the first occurrence of a factor u of x.

DEFINITION 1. For $u \in Fact(x)$
$foccur(u, x) = \min\{|z| \mid z = wu \text{ and } x = wuv\}$.

In words, $foccur(u, x)$ is the ending position of the first occurrence of u in x.

The algorithm FACTOR-ORACLE-OFF-LINE(x, m) presented in Figure 1 builds the factor oracle of a string x of length m.

DEFINITION 2. We call factor oracle of x the automaton obtained by algorithm the FACTOR-ORACLE-OFF-LINE(x, m) where all the states are terminal. It is denoted by $\mathcal{O}(x)$.

The factor oracle of a string x of length m is a Deterministic Finite Automaton (Q, q_0, F, δ) where $Q = \{0, 1, \ldots, m\}$ is the set of states, $q_0 = 0$

```
FACTOR-ORACLE-OFF-LINE(x, m)
 1 for i ← 0 to m do
 2     create state i
 3 for i ← 0 to m − 1 do
 4     δ(i, x[i + 1]) ← i + 1
 5 for i ← 0 to m − 1 do
 6     u ← string of minimal length recognized at state i
 7     for each a ∈ Σ with a ≠ x[i + 1] do
 8         if ua ∈ Fact(x[i − |u| + 1..m]) then
 9             δ(i, a) ← i + foccur(ua, x[i − |u| + 1..m])
```

Figure 1. Algorithm for off-line construction of the factor oracle of x.

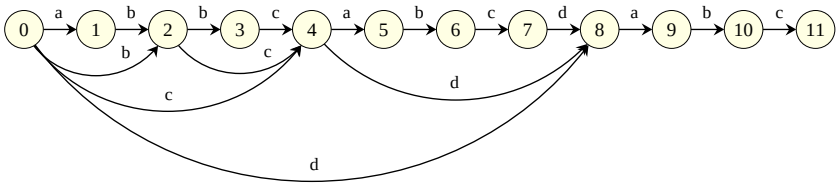

Figure 2. Factor oracle of $x = abbcabcdabc$. All the states are terminal. All the factors of x are recognized. Some more strings that are not factors but subsequences of x, such as $abca$, are also recognized.

is the starting state, $T = Q$ is the set of terminal states and δ is the transition function. An example of factor oracle is given in Figure 2. The factor oracle of a string x of length m has the following properties: it has exactly $m + 1$ states (there is a bijection between the states and the length of all the prefixes of x, including the empty one). Furthermore it can be shown that its number of transitions is linear.

LEMMA 3 ([1]). *The factor oracle of a string x of length m, $\mathcal{O}(x)$, has between m and $2m - 1$ transitions.*

The automaton $\mathcal{O}(x)$ recognizes at least all the factors of x. Actually its language includes the set of all the factors of x and is included in the set of all the subsequences of x.

Construction method from the suffixes

This method, introduced in [6], is based on the suffixes of x.

```
FACTOR-ORACLE-FROM-SUFFIXES(x, m)
 1  for i ← 0 to m do
 2     create state i
 3  for i ← 0 to m − 1 do
 4     δ(i, x[i + 1]) ← i + 1
 5  for i ← 2 to m do
 6     j ← 0
 7     k ← i − 1
 8     while δ(j, x[k + 1]) ≠ UNDEFINED do
 9        j ← δ(j, x[k + 1])
10        k ← k + 1
11     if k ≠ m then
12        δ(j, x[k + 1]) ← k + 1
```

Figure 3. Algorithm building the factor oracle of the string x of length m from the suffixes of x.

It consists in constructing the skeleton of the factor oracle of x. The skeleton is actually the set of states i and the transitions from state $i-1$ to state i labeled by $x[i]$ with $1 \leq i \leq m$. Then, each state i from 2 to m is scanned. The algorithm looks for the longest prefix of the suffix of x, starting in position i, which is already recognized by the current automaton. If such a string $x[i..k]$ (with $k \neq m$) is recognized in state j then a transition is created from j to $k+1$ labeled by $x[k+1]$ (k is equal to $i-1$ if there exists no such prefix).

The algorithm implementing this idea can be seen in Figure 3 and an example is shown in Figure 4.

3.2 On-line construction

We now introduce an on-line method for constructing the factor oracle of a string. For that we first give the definition of the longest repeated suffix of the prefix of length i of m.

DEFINITION 4. $LRS(i, x)$ is the longest suffix of $x[1..i]$ having two occurrences in $x[1..i]$.

We can now give the definition of the suffix link of a state.

DEFINITION 5. For $1 \leq i \leq m$, $S(i, x) = \delta(0, LRS(i, x))$ and $S(0, x) = -1$.

The suffix link of a state i, with $1 \leq i \leq m$, is the state where the longest repeated suffix of $x[1..i]$ is recognized. These suffix links are essential tools

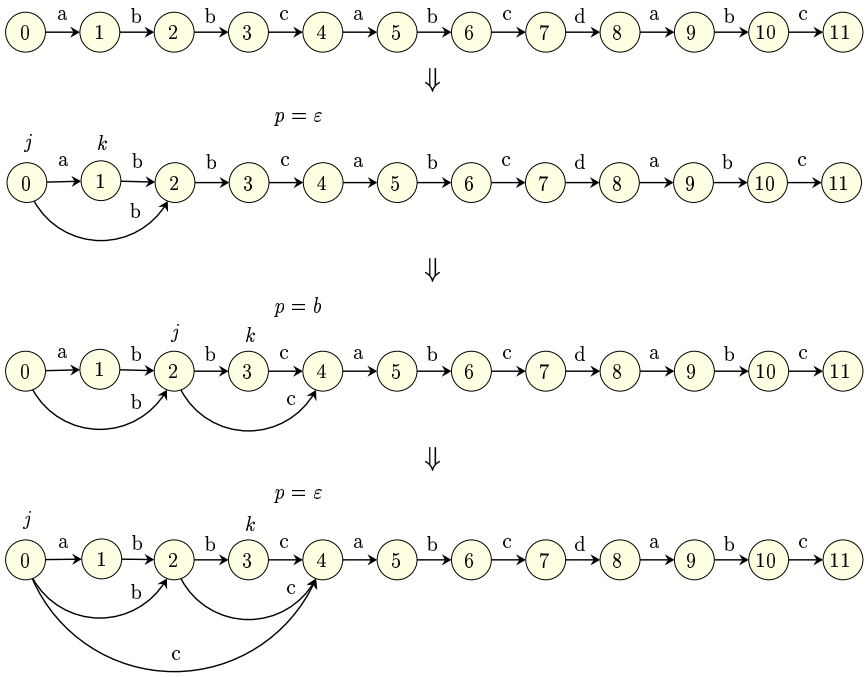

Figure 4. Construction from suffixes of the factor oracle of $x = abbcabcdabc$ step by step. Only the first four steps are represented. The first step consists in computing the skeleton of the factor oracle. Then, during each step, the values of j, k and the longest recognized prefix p of $x[i..m]$ are given.

```
FACTOR-ORACLE-ON-LINE(x, m)
 1  create state 0
 2  S[0] ← −1
 3  for i ← 1 to m do
 4     create state i
 5     δ(i − 1, x[i]) ← i
 6     s ← S[i − 1]
 7     while s > −1 and δ(s, x[i]) is not defined do
 8        δ(s, x[i]) ← i
 9        s ← S[s]
10     if s = −1 then
11        S[i] ← 0
12     else S[i] ← δ(s, x[i])
```

Figure 5. Algorithm for on-line construction of the factor oracle of x.

for designing an on-line construction of $\mathcal{O}(x)$.

We now define the suffix path of a state.

DEFINITION 6. For $0 \leq i \leq m$, $SP(i,x) = (k_0 = i, k_1, \ldots, k_\ell = 0)$ such that $k_j = S(k_{j-1}, x)$ for $1 \leq j \leq \ell$.

The algorithm FACTOR-ORACLE-ON-LINE(x,m) given in Figure 5 presents an on-line construction of the factor oracle of x where suffix links are implemented by a table S of size $m+1$ ($S[i] = S(i,x)$ for $0 \leq i \leq m$). The oracle $\mathcal{O}(x[1..i])$ can be obtained from $\mathcal{O}(x[1..i-1])$ by adding a transition labeled by $x[i]$ to state i from all the states of $SP(i-1,x)$ for which there does not already exist an outgoing transition labeled by $x[i]$. The suffix link of state i is then set with the largest state of $SP(i-1,x)$ for which an outgoing transition labeled by $x[i]$ already existed (or with the initial state if no such state is found).

The on-line construction of the factor oracle is linear.

THEOREM 7 ([1]). *The complexity of the algorithm* FACTOR-ORACLE-ON-LINE(x,m) *is* $O(m)$ *in time and space.*

The factor oracle of x is acyclic: all the transitions are going from a state i to a state j with $0 \leq i < j \leq m$. Furthermore the automaton $\mathcal{O}(x)$ is homogeneous: all the transitions leading to a state are labeled by the same letter, more precisely, all the transitions going to state i are labeled by $x[i]$, for $1 \leq i \leq m$. We distinguish two kinds of transitions: transitions from state i to state $i+1$ are called *internal transitions* and transitions from

```
FACTOR-ORACLE(x, m)
 1  E ← ∅
 2  S[0] ← −1
 3  for i ← 1 to m do
 4     s ← S[i − 1]
 5     while NEXT-STATE(s, x[i]) = UNDEFINED do
 6        E ← E ∪ {(s, i)}
 7        s ← S[s]
 8     S[i] ← NEXT-STATE(s, x[i])
```

Figure 6. Algorithm for on-line construction of the factor oracle of x without storing the states, the internal transitions and the labels of the external transitions.

```
NEXT-STATE(s, a)
 1  if s = −1 then
 2     return 0
 3  else if a = x[s + 1] then
 4     return s + 1
 5  else if ∃(s, j) ∈ E such that x[j] = a then
 6     return j
 7  else return UNDEFINED
```

Figure 7. Algorithm that gives the target state of the transition from state s labeled by a or UNDEFINED if there is no such transition. It considers an artificial state -1 such that $S(0, x) = -1$ and $\delta(-1, a) = 0$ for all $a \in \Sigma$.

state i to state j such that $j - i > 1$ are called *external transitions*. There are exactly m internal transitions. Thus, to store the oracle, one needs to store only the string x and at most $m - 1$ external transitions without their label. All the other information can be deduced from the string x. This representation is completely independent of the underlying alphabet.

The algorithm FACTOR-ORACLE(x, m) given in Figure 6 builds the factor oracle without creating the states and the internal transitions. The external transitions are stored in a set denoted by E. It uses an algorithm NEXT-STATE(s, a), given in Figure 7, that gives the target of the transition from state s labeled by a or UNDEFINED if there is no such transition. It considers an artificial state -1 such that $S(0, x) = -1$ and $\delta(-1, a) = 0$ for all $a \in \Sigma$.

An example of such a factor oracle is given in Figure 8.

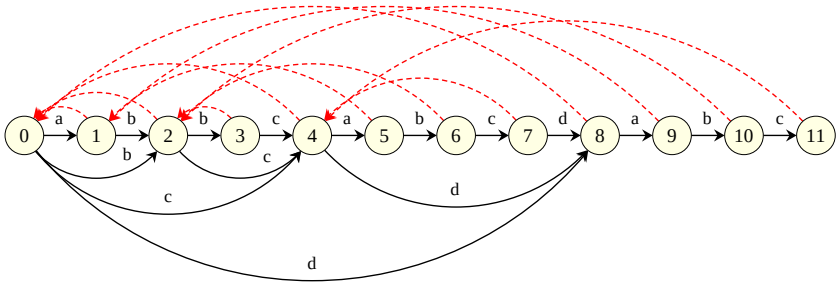

Figure 8. Factor oracle of $x = abbcabcdabc$. Dash arrows represent the suffix links. All transitions leading to state i are labeled by $x[i]$. There is always a transition from state $i-1$ to state i, for $1 \leq i \leq m$. Such transitions are called internal transitions. Thus this factor oracle can be represented by x itself and the list $((0,2),(0,4),(0,8),(2,4),(4,8))$ of the external transitions (going from a state i to a state j such that $j - i > 1$ with $0 \leq i \leq m-1$ and $1 \leq j \leq m$) without their label. All the states are terminal. All the factors of x are recognized. Some more strings that are not factors but subsequences of x, such as $abca$, are also recognized.

```
SUFFIX-ORACLE(x, m)
 1  FACTOR-ORACLE(x, m)
 2  T ← ∅
 3  t ← m
 4  while S[t] ≠ −1 do
 5     T ← T ∪ {t}
 6     t ← S[t]
```

Figure 9. Algorithm for on-line construction of the suffix oracle of x.

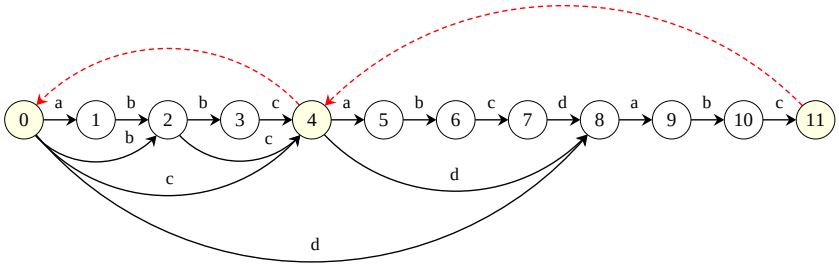

Figure 10. Suffix oracle of $x = abbcabcdabc$. Only the states of $SP(11, abbcabcdabc)$ are terminal: 0, 4 and 11.

4 Suffix oracle

The suffix oracle of a string x of length m recognizes at least all the suffixes of x.

A state i of the suffix oracle of x is terminal if there exists a path labeled by a suffix of x from the initial state 0 to state i.

The suffix oracle of a string x of length m is an automaton similar to the factor oracle of x where only the states of $SP(m, x)$ are terminal.

The algorithm SUFFIX-ORACLE(x, m) given in Figure 9 implements this idea. An example of suffix oracle is given in Figure 10.

5 String-matching

String-matching consists in finding one, or more generally, all the occurrences of a string x of length m (usually called a *pattern*) in a string y of length n (more usually called a *text*).

Applications require two kinds of solutions depending on which string,

the pattern or the text, is given first. Algorithms based on the use of automata or combinatorial properties of strings are commonly implemented to preprocess the pattern and solve the first kind of problem. The notion of indexes realized by trees or automata is used in the second kind of solutions. The string-matching algorithms presented in this section solve the first kind of problem.

Most string-matching algorithms work as follows. They scan the text with the help of a window of size m. They first align the left ends of the window and the text, then compare the characters of the window with the characters of the pattern – this specific work is called an *attempt* – and after a whole match of the pattern or after a mismatch they *shift* the window to the right. They repeat the same procedure again until the right end of the window goes beyond the right end of the text. This mechanism is usually called the *sliding window mechanism*. Each attempt is associated with the position j in the text when the window is positioned on $y[j..j+m-1]$. See [4] for an introduction on this subject.

5.1 Backward oracle matching

The Boyer-Moore type algorithms (see [4] and [21]), that are among the most efficient in practice, match some suffixes of the pattern but it is possible to match some prefixes of the pattern when scanning the character of the window from right to left and then improve the length of the shifts. One possibility consists in using the factor oracle of the reverse pattern. The string-matching algorithm using the factor oracle of the reverse pattern is called the Backward Oracle Matching (BOM) algorithm [1, 2].

The preprocessing phase of the Backward Oracle Matching algorithm consists in computing the factor oracle for the reverse pattern x^R. Despite the fact that it is able to recognize strings that are not factor of the pattern, the factor oracle can be used to do string-matching since the only string of length greater or equal m which is recognized by the oracle is the reverse pattern itself.

During the searching phase the Backward Oracle Matching algorithm parses the characters of the window from right to left with the automaton $\mathcal{O}(x^R)$ starting with the initial state 0. It goes until there is no more transition defined for the current character. At this time it means either that the suffix of the window cannot be a factor of the pattern or that an occurrence of the pattern has been found. In the latter case the algorithm applies a shift of length 1. In the first case the Backward Oracle Matching algorithm applies a shift that aligns the left end of the window with the last but one scanned character of the text (see Figure 11).

The Backward Oracle Matching algorithm is shown in Figure 12.

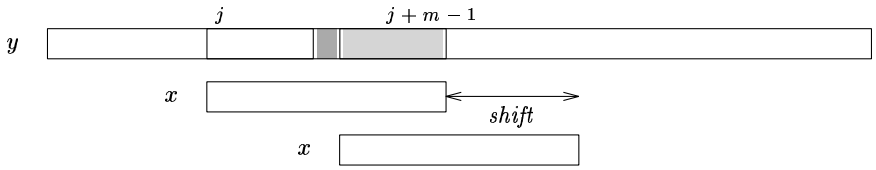

Figure 11. General situation during the BOM algorithm: the window is positioned on $y[j..j+m-1]$, a suffix of $y[j..j+m-1]$ has successfully been parsed with the factor oracle of x^R (light gray) and a transition is not defined from the current state labeled by a character of y (dark gray); the shift consists in bringing the window just after this character.

```
BOM(x, m, y, n)
 1  FACTOR-ORACLE(x^R, m)
 2  j ← 0
 3  while j ≤ n − m do
 4     s ← 0
 5     i ← m
 6     while NEXT-STATE(s, y[i + j]) ≠ UNDEFINED do
 7        s ← NEXT-STATE(s, y[i + j])
 8        i ← i − 1
 9     if i = 0 then
10        OUTPUT(j + 1)
11        i ← 1
12     j ← i + j
```

Figure 12. The Backward Oracle Matching algorithm.

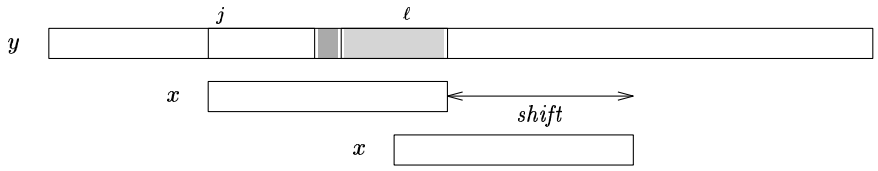

Figure 13. General situation during the BSOM algorithm: the window is positioned on $y[j..j+m-1]$, a suffix of $y[j..j+m-1]$ has successfully been parsed with the suffix oracle of x^R (light gray) and a transition is not defined from the current state labeled by a character of y (dark gray); the last terminal state has been encountered when scanning $y[\ell]$; the shift consists in bringing the left end of the window just on this character.

5.2 Backward suffix oracle matching

The preprocessing phase of the Backward Suffix Oracle Matching (BSOM) algorithm [1, 2] consists in computing the suffix oracle for the reverse pattern x^R. The searching phase is similar to the one of the BOM algorithm except for computing the shift lengths. When an attempt is terminated, the length of the longest prefix of the pattern which is a suffix of the scanned part of the text is less than the length of the path taken in $\mathcal{O}(x^R)$ from the initial state 0 and the last terminal state encountered. Knowing this length, it is trivial to compute the length of the shift to perform (see Figure 13).

The Backward Suffix Oracle Matching algorithm is shown in Figure 14.

5.3 Turbo-BOM algorithm

The algorithms BOM and BSOM are quadratic in the worst case. It is possible to make them linear in terms of text character inspections and comparisons by using the Knuth-Morris-Pratt (KMP) algorithm [12] in addition to the oracle.

The KMP algorithm is one of the most famous exact string-matching algorithms (see [4]). During an attempt it scans the characters of the window from left to right. In case of the mismatch or of an occurrence of the pattern in the text, it uses a precomputed table to compute the length of the shift. For $1 \leq i \leq m$, we define $kmpNext[i]$ as being the longest border of $x[1..i-1]$ followed by a character different from $x[i]$. The value of $kmpNext[m]$ is set with the period of x. During an attempt of the KMP algorithm at position j, when a mismatch occurs at position $j+k$ (with $0 \leq k \leq m$), a shift of length $m - kmpNext[k-1]$ is applied and the comparisons are resumed with $y[j+k]$ and $x[kmpNext[k-1]]$. The values of the table $kmpNext$ can be computed in $O(m)$ (see [12, 4]).

A general situation of the Turbo-BOM algorithm [2] is the following.

```
BSOM(x, m, y, n)
 1  SUFFIX-ORACLE(x^R, m)
 2  j ← 0
 3  while j ≤ n − m do
 4     s ← 0
 5     i ← m
 6     shift ← m
 7     while NEXT-STATE(s, y[i + j]) ≠ UNDEFINED do
 8        s ← NEXT-STATE(s, y[i + j])
 9        if s ∈ T then
10           period ← shift
11           shift ← i
12        i ← i − 1
13     if i = 0 then
14        OUTPUT(j + 1)
15        shift ← period
16     j ← j + shift
```

Figure 14. The Backward Suffix Oracle Matching algorithm.

The window is positioned on the text factor $y[j..j+m-1]$. A prefix $u = y[j..j+|u|-1]$ has been matched using the KMP algorithm. The position $j+|u|-1$ is called a critical position and is denoted by *critpos*. The scan with the factor oracle starts with the position $j+m-1$ and goes leftward. Two cases arise whether the critical position is reached or not:

1. The critical position is not reached. A mismatch occurs at a position k such that $critpos < k \leq j+m-1$. The window is shifted and positioned on the text factor $y[k+1..k+m]$ (see Figure 15). The search is resumed with the KMP algorithm at position $k+1$, between $x[1]$ and $y[k+1]$, going rightward, rescanning at least the text characters scanned with the factor oracle up to the text position $k+m$ and then stops when a recognized prefix of x in y is small enough (less than αm with $0 < \alpha < 1$, experimentally $\alpha = 1/2$ seems to be a good value).

2. The critical position is reached. The search is then resumed using the KMP algorithm at position $critpos+1$, between $x[|u|+1]$ and $y[critpos+1]$, going rightward, rescanning at least the text characters scanned with the factor oracle up to position $k+m$ and then stops when a recognized prefix of x in y is small enough (less than αm).

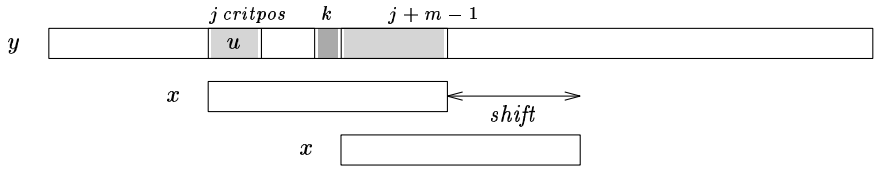

Figure 15. General situation during the Turbo-BOM algorithm: the window is positioned on $y[j..j+m-1]$, a prefix u of x has been previously matched by the KMP algorithm, a suffix of $y[j..j+m-1]$ has successfully been parsed with the factor oracle of x^R (light gray) and a transition is not defined from the current state labeled by $y[k]$ (dark gray); the shift consists in bringing the window just after this character.

The first attempt scans with the factor oracle from $y[m]$ going leftward. The occurrences of x in y are always reported when using the KMP algorithm. The algorithm TURBO-BOM(x, m, y, n, α) is given in Figure 16. It uses an algorithm KMPCOMPARE(i', j'), given in Figure 17, that compares the two characters $x[i']$ and $y[j']$.

5.4 Turbo-BSOM algorithm

The Turbo-BSOM algorithm [2] is similar to the Turbo-BOM algorithm except that it uses the suffix oracle of x^R rather than the factor oracle. The difference comes from the computation of the shift when in case 1 the critical position is not reached: a mismatch occurs at a position k such that $critpos < k \leq j + m - 1$ and the last terminal state was encountered when scanning $y[\ell]$ with $k < \ell \leq j + m - 1$. The window is shifted and positioned on the text factor $y[\ell..\ell + m - 1]$ (see Figure **??**). The search is resumed with the KMP algorithm at position ℓ, going rightward, rescanning at least the text characters scanned with the factor oracle up to position $k + m$ and then stops when a recognized prefix of x in y is small enough (less than αm with $0 < \alpha < 1$, experimentally $\alpha = 1/2$ seems to be a good value).

The algorithm TURBO-BSOM(x, m, y, n, α) is given in Figure 19.

5.5 Complexities

The Backward Oracle Matching and Backward Suffix Oracle Matching algorithms have a quadratic worst case time complexity but it is conjectured to be optimal in the average [2]. It is supposed to perform $O(n \log_{|\Sigma|} m/m)$ inspections of text characters reaching the best bound shown by Yao in 1979 [26]. The reader can refer to [21] for an interesting study about the practical behavior of the BOM algorithm.

It can easily be shown that the algorithms TURBO-BOM and TURBO-

```
TURBO-BOM(x, m, y, n, α)
 1  FACTOR-ORACLE(x^R, m)
 2  critpos ← 0
 3  j ← 0
 4  while j ≤ n − m do
        ▷ Scanning with the factor oracle
 5     s ← 0
 6     i ← m
 7     while i + j > critpos and
           NEXT-STATE(s, y[i + j]) ≠ UNDEFINED do
 8        s ← NEXT-STATE(s, y[i + j])
 9        i ← i − 1
10     if i + j = critpos then
11        i' ← critpos + 1
12        j' ← j
13     else i' ← 1
14        j' ← i + j + 1
        ▷ Rescanning with KMP
15     while j' ≤ j + m − 1 do
16        (i', j') ← KMPCOMPARE(i', j')
        ▷ Scanning with KMP
17     while j' ≤ n and i' > αm do
18        (i', j') ← KMPCOMPARE(i', j')
19     critpos ← i' − 1
20     j ← j' − i' − 1
```

Figure 16. The Turbo-BOM algorithm.

```
KMPCOMPARE(i', j')
1  while i' > 0 and x[i'] = y[j'] do
2     i' ← kmpNext[i']
3  i' ← i' + 1
4  j' ← j' + 1
5  if i' > m then
6     OUTPUT(j' − i')
7     i' ← kmpNext[i']
8  return (i', j')
```

Figure 17. Comparisons of $x[i']$ and $y[j']$ with the KMP algorithm.

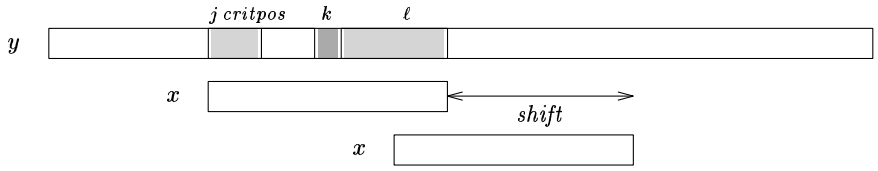

Figure 18. General situation during the Turbo-BSOM algorithm: the window is positioned on $y[j..j+m-1]$, a suffix of $y[j..j+m-1]$ has successfully been parsed with the suffix oracle of x^R (light gray) and a transition is not defined from the current state labeled by a character of y (dark gray); the last terminal state has been encountered when scanning $y[\ell]$; the shift consists in bringing the left end of the window just on this character.

BSOM are linear in the worst case.

THEOREM 8 ([2]). *When searching a pattern in a text of length n, 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 $O(\log |\Sigma|)$.*

The reader can refer to [2] for a discussion about the choice of value for the parameter α.

6 Detection of repeats in large genomic sequences

One of the numerous areas of study in bioinformatics concerns the problem of repeat detection in long DNA sequences. A DNA sequence can be seen as a string written on the four letter alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ and a repeat is a factor of this sequence that occurs in at least two positions. Several theoretical studies have been realized on this subject (see [23, 7, 24]). The main problem is the size of the sequences (from a few letters to hundreds of millions of letters): detecting all the repeats (of each length and for each position) in these sequences becomes a very difficult problem in practice. To solve this problem, different approaches have been used such as hashing techniques (see [25]), data compression algorithms (see [22, 5]) or efficient implementations of data structures like suffix trees (see [14]).

In this section, we present a repeat detection method based on the factor oracle of a sequence. These results and their applications have been published in [16] and [19].

```
TURBO-BSOM(x, m, y, n, α)
 1  SUFFIX-ORACLE(x^R, m)
 2  critpos ← 0
 3  j ← 0
 4  while j ≤ n − m do
        ▷ Scanning with the suffix oracle
 5     s ← 0
 6     i ← m
 7     shift ← m
 8     while i + j > critpos and
           NEXT-STATE(s, y[i + j]) ≠ UNDEFINED do
 9        s ← NEXT-STATE(s, y[i + j])
10        if s ∈ T then
11           shift ← i
12        i ← i − 1
13     if i + j = critpos then
14        i' ← critpos + 1
15        j' ← j
16     else i' ← 1
17        j' ← j + shift + 1
        ▷ Rescanning with KMP
18     while j' ≤ j + m − 1 do
19        (i', j') ← KMPCOMPARE(i', j')
        ▷ Scanning with KMP
20     while j' ≤ n and i' > αm do
21        (i', j') ← KMPCOMPARE(i', j')
22     critpos ← i' − 1
23     j ← j' − i' − 1
```

Figure 19. The Turbo-BSOM algorithm.

6.1 Definitions

The suffix links in the factor oracle indicates a repeat: for a position i on x, if $S(i,x)$ is different from 0 then a suffix of $x[1..S(i,x)]$ is identical to a suffix of $x[1..i]$. The difficulty is to compute the length of this suffix. We will first formalize this notion.

DEFINITION 9. Given a string x of length m and its factor oracle $\mathcal{O}(x)$, $\textsc{Rep}(i,x)$, for $0 \leq i \leq m$, is equal to ua where u is one of the longest repeated suffix of $x[1..i]$ ending in the position equal to the greatest element of $SP(S(i,x)-1,x) \cap SP(i-1,x)$, and $a = x[i]$.

The following method computes $|\textsc{Rep}(i,x)|$ (denoted by $lrs(i,x)$ for the *length of a repeated suffix*) for $1 \leq i \leq m$. For that it looks for the length of a common suffix of $x[1..SP(i,x)]$ and $x[1..i]$. Looking for this common suffix is equivalent to look for a common suffix of $x[1..SP(i,x)-1]$ and $x[1..i-1]$, since $x[i] = x[SP(i,x)]$. The following lemma formalizes this notion.

LEMMA 10 ([16]). *$|\textsc{Rep}(i,x)| = lrs(i,x)$, for $0 \leq i \leq m$, is equal to the length of a common suffix of $x[1..SP(i,x)-1]$ and $x[1..i-1]$, plus 1.*

In order to compute the length of each $\textsc{Rep}(i,x)$, one needs to find efficiently the greatest element of $SP(S(i,x)-1,x) \cap SP(i-1,x)$. Consider the two states π_1 and π_2 defined as follows.

DEFINITION 11. Let j denote the greatest element of $SP(S(i,x)-1,x) \cap SP(i-1,x)$. π_1 is equal to the state of $SP(i-1,x)$ such that $S(\pi_1,x) = j$. π_2 is equal to the state of $SP(S(i,x)-1,x)$ such that $S(\pi_2,x) = j$, if $S(i,x)-1 \neq j$, or is equal to j if $S(i,x)-1 = j$.

This definition is illustrated in Figure 20.

All these definitions lead to the definition of the *lrs* values.

DEFINITION 12. Consider that $\mathcal{O}(x)$ is built. Then $lrs(0,x) = 0$ and $\forall i$, $0 < i \leq m$:

$$lrs(i,x) = \begin{cases} 0 & \text{if } S(i,x) = 0 \quad (1) \\ lrs(\pi_1,x)+1 & \text{if } \pi_2 = S(\pi_1,x) \quad (2) \\ \min(lrs(\pi_1,x), lrs(\pi_2,x))+1 & \text{otherwise} \quad (3) \end{cases}$$

6.2 Algorithms

The algorithm FACTOR-ORACLE-WITH-LRS(x,m), Figure 21, computes the factor oracle of x and the values *lrs* according to the definitions given above. It uses the algorithm LENGTH-REPEATED-SUFFIX shown in Figure 22. The values *lrs* are implemented by a table of size $m+1$ ($lrs[i] = lrs(i,x)$ for $0 \leq i \leq m$).

We can now present the main result of this section.

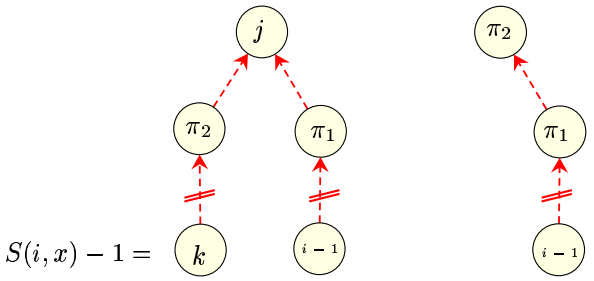

Figure 20. Definition of π_1 and π_2. Dashed arrows are suffix links. Interrupted dashed arrows represent a series of suffix links. On the left, $\pi_2 \in SP(i,x)-1$ (denoted by k) and $\pi_1 \in SP(i-1,x)$ have the same suffix link j. On the right, a particular case: π_2 is equal to $S(\pi_1,x)$.

```
FACTOR-ORACLE-WITH-LRS(x, m)
 1  E ← ∅
 2  S[0] ← −1
 3  lrs[0] ← 0
 4  for i ← 1 to m do
 5     s ← S[i − 1]
 6     π1 ← i − 1
 7     while NEXT-STATE(s, x[i]) = UNDEFINED do
 8        E ← E ∪ {(s, i)}
 9        π1 ← s
10        s ← S[s]
11     S[i] ← NEXT-STATE(s, x[i])
12     lrs[i] ← LENGTH-REPEATED-SUFFIX(π1, s)
```

Figure 21. Algorithm for on-line construction of the factor oracle of x and the computation of the *lrs* values.

```
LENGTH-REPEATED-SUFFIX(π1, s)
 1 if s = 0 then
 2    return 0
 3 else π2 ← s − 1
 4    if π2 = S[π1] then
 5       return lrs[π1] + 1
 6    else while S[π2] ≠ S[π1] do
 7            π2 ← S[π2]
 8 return min(lrs[π1], lrs[π2]) + 1
```

Figure 22. Algorithm computing the *lrs* values, called by the algorithm FACTOR-ORACLE-WITH-LRS.

THEOREM 13 ([16]). *The algorithm* FACTOR-ORACLE-WITH-LRS(x, m) *computes the factor oracle of the word* x *of length* m *and computes* $lrs(i, x)$ *for* $0 \leq i \leq m$*, in linear time and space.*

6.3 Applications

The above method has been implemented and reveals very fast in practice. For instance it processes chromosome II of the model plant *Arabidopsis thaliana* (which length is about 19.6 millions of letters) in 32.7 seconds on a computer with 1Go memory and a 500MHz processor [19].

However the method does not detect the length of the longest repeated suffix for about 40% of the positions of the studied sequence. But the average difference between this length and the computed value is close to 1. An improvement of this method enables to detect the correct value for about 98% of the positions with an average difference (between the length of the longest suffix and the computed value) close to 0.5. The main drawback of this improvement is that it doubles the space requirement of the data structure [18] (see [11] for recent developments).

The *lrs* values only detect exact repeats which is not completely satisfactory when studying genomic sequences where substitutions, deletions or insertions are known to occur frequently. Those exact repeats can serve as seeds to find approximate repeats using a classical extension technique (as in BLAST [13] for instance). Detecting approximate repeats with a threshold of 85% of similarity by extending exact repeats of length at least 15 only doubles the running times [19].

A software named `FORRepeats` [19] have been developed based on the above method. This method has been used for detecting repeats in whole chromosomes of various organisms. By concatenating two sequences, sep-

arated by a joker (letter that is not an element of the alphabet of the two sequences) and by considering only suffix links crossing the boundary between the two sequences, it is possible to compare two sequences [19].

7 Linear on-line lossless data compression

Another area of research of great interest in computer science is data compression. A lot of different data compression techniques have been developed based on different approaches. Some of them are well-known such as factorization techniques (see [27]) or Burrows-Wheeler transform (see [8]). The technique we present here take place in the factorization methods.

We introduce a new factorization method based on the computation of the factor oracle of the data to be compressed and the *lrs* values presented in the previous section.

7.1 The o-factorization

First, let us recall the definition of a factorization of a string.

DEFINITION 14. The factorization of a string x of length m is equal to a series of factors $(u_1, u_2, .., u_k)$ such that $x = u_1 u_2 \cdots u_k$.

First introduced and described in [15] the o-factorization is defined as follows.

DEFINITION 15. Consider the factor oracle of a string x of length m and the corresponding *lrs* values computed. The o-factorization of x, denoted by $o\mathcal{F}(x, m)$, is equal to the series of factors $(u_1, u_2, .., u_k)$ such that:

- $u_1 = x[1]$;
- if the first j factors of $o\mathcal{F}(x, m)$, with $|u_1..u_j| = \ell$, are defined then:
  - $u_{j+1} = x[\ell + 1]$ if $lrs(\ell + 1, x) = 0$;
  - $u_{j+1} = x[\ell + 1..i - 1]$ with $\ell + 1 < i \leq m$, $lrs(i - 1, x) \geq i - 1 - \ell$ and $(i = m + 1$ or $i - \ell > lrs(i, x))$, otherwise.

Figure 23 illustrates the different cases of this definition.

Now that the o-factorization of a string x of length m is given, we can present the algorithm computing this factorization and consequently encoding x.

7.2 The encoding model

In the o-factorization of a string, when a factor does not consist in a single letter, it means that it has a copy on the left, this copy can be encoded by a pair (starting position, length).

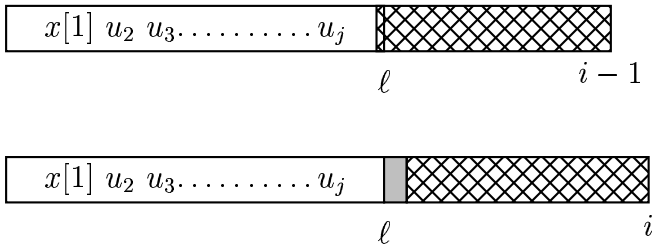

Figure 23. Illustration of the definition of the o-factorization. The factorization of the prefix of length ℓ has already been computed. The dashed part represent the length of the repeated suffixes computed in positions $i-1$ and i (given by $lrs[i-1]$ and $lrs[i]$). The value of $lrs[i-1]$ is great enough to cover the distance between $i-1$ and ℓ: the repeated factor in positions $i-1$ and $S[i-1]$ of length $i-1-\ell$ is a good candidate for being the next factor in the o-factorization. If the value of $lrs[i]$ is less than the distance between positions i and ℓ: it means that the repeated factor detected in position i is too short and can not be a good candidate for being the next factor in the o-factorization. Thus $u_{j+1} = x[\ell+1..i-1]$.

The algorithm ENCODING-WITH-FACTOR-ORACLE(x, m) given in Figure 24 encodes a string x of length m, according to the definition 15,

This algorithm is based on the algorithm FACTOR-ORACLE-WITH-LRS. The variable ℓ represents the last position in which a compression has been performed (*i.e.* ℓ is the length of the prefix already encoded). Each time $lrs[i]$ is computed, the algorithm tests if it is long enough. Two cases can appear:

- if $lrs[i] \geq i - \ell$ (line 13), then:
  - if ℓ is strictly less than $i - 1$, it means that a factor starting in position $S[i-1] - (i - 1 - \ell) + 1$ of length $i - 1 - \ell$ has to be encoded,
  - and, if $lrs[i] = 0$ a new letter has to be encoded.
- otherwise nothing is done.

The last two lines prevent from a border effect. If the last encoded factor is not the last factor of the $o\mathcal{F}$, then this last factor is encoded.

Since the o-factorization of a string is unique and can be computed in linear time and space, the following theorem holds.

THEOREM 16. *For a string x of length m, the algorithm* ENCODING-WITH-FACTOR-ORACLE *encodes x in linear time and space.*

The decoding of a sequence obtained by the algorithm ENCODING-WITH-FACTOR-ORACLE(x, m) is immediate.

7.3 Applications

A software named `compror` as been implemented using the above compression model [17]. It uses auto-delimited codes, namely Fibonacci codes of order 2 and 3, for representing integers. It gives very good results for compressing archive files containing PostScript and Encapsulated PostScript documents, even beating `bzip2`.

8 Conclusion and open problems

We have presented several applications of the factor oracle. Recently this data structure has been used for text indexing [9, 10], finding maximal repeats [11] and music analysis [3]. New properties of this structure can be found in [20].

The main open problem concerning the factor oracle of a string x is to precisely characterized the language recognized by the oracle. It would probably help to recognized on-line if a path in the oracle corresponds or not to a factor of x. The on-line construction gives a method to build the

```
ENCODING-WITH-FACTOR-ORACLE(x, m)
 1  E ← ∅
 2  S[0] ← −1
 3  lrs[0] ← 0
 4  ℓ ← 0
 5  for i ← 1 to m do
 6     s ← S[i − 1]
 7     π1 ← i − 1
 8     while NEXT-STATE(s, x[i]) = UNDEFINED do
 9        E ← E ∪ {(s, i)}
10        π1 ← s
11        s ← S[s]
12     S[i] ← NEXT-STATE(s, x[i])
13     lrs[i] ← LENGTH-REPEATED-SUFFIX(π1, s)
13     if lrs[i] < i − ℓ then
14        if ℓ ≠ i − 1 then
             ▷ The factor is x[S[i − 1] − (i − 1 − ℓ) + 1]..S[i − 1]]
15           CODING-FACTOR(S[i − 1] − (i − 1 − ℓ) + 1, i − 1 − ℓ)
             ▷ The length of the compressed prefix is now ℓ = i − 1
16           ℓ ← i − 1
17        if lrs[i] = 0 then
             ▷ The factor is x[i]
18           CODING-LETTER(x[i])
             ▷ The length of the compressed prefix is now ℓ = i
19           ℓ ← i
20  if ℓ < m then
       ▷ The last coded factor is x[S[m] − (m − ℓ) + 1..S[m]]
21     CODING-FACTOR(S[m] − (m − ℓ) + 1, m − ℓ)
```

Figure 24. Data compression algorithm with a factor oracle. Notations correspond to those used in definition 15. CODING-LETTER and CODING-FACTOR algorithms are not given since they depend on the techniques one wants to use to encode each factor of the $o\mathcal{F}$.

factor oracle of xa when the factor oracle of x is given. Is there an efficient method for constructing the oracle of x when the oracle of ax is given? Is there an algorithm in $O(k)$ time for building the oracle of $w[i+1..i+k]$ when the oracle of $w[1..i]$ is given?

BIBLIOGRAPHY

[1] C. Allauzen, M. Crochemore, M. Raffinot, Factor oracle: a new structure for pattern matching, In J. Pavelka, G. Tel, and M. Bartosek, editors, *SOFSEM'99, Theory and Practice of Informatics*, LNCS 1725, pages 291–306, Milovy, Czech Republic, 1999.

[2] C. Allauzen, M. Crochemore, M. Raffinot, Efficient Experimental String Matching by Weak Factor Recognition, *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, LNCS 2089, pages 51–72, Springer-Verlag, Berlin, 2001.

[3] G. Assayag, S. Dubnov, Using factor oracles for machine improvisation, *Soft Computing*, to appear.

[4] C. Charras, T. Lecroq, *Handbook of exact string matching algorithms*, King's College Publications, 2004.

[5] X. Chen, S. Kwong, M. Li, A compression algorithm for DNA sequences and its applications in genome comparison, *In Proceedings of the 4th Annual International Conference on Computational Molecular Biology*, 107, 2000.

[6] L. Cleophas, G. Zwaan, B. W. Watson, Constructing factor oracles. *In Proceedings of the Prague Stringology Conference*, 37–50, 2003.

[7] M. Crochemore, W. Rytter, *Jewels of Stringology*, World Scientific, 2002.

[8] P. Fenwick, The Burrows-Wheeler transform for block sorting text compression – principles and improvements, *The Computer Journal*, 39(9):731–740, 1996.

[9] R. Kato, A New Text Search Algorithm Using Factor Oracle as Full-Text Index, Technical Report TR-C179, Dept. of Math. and Comp. Sciences, Tokyo Institute of Technology, 2003.

[10] R. Kato, A New Full-Text Search Algorithm Using Factor Oracle as Index, Technical Report TR-C185, Dept. of Math. and Comp. Sciences, Tokyo Institute of Technology, 2003.

[11] R. Kato, Finding Maximal Repeats with Factor Oracles, Technical Report TR-C190, Dept. of Math. and Comp. Sciences, Tokyo Institute of Technology, 2004.

[12] D. E. Knuth, J. H. Morris, Jr and V. R. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* **6**(1) (1977) 323–350.

[13] I. Korf, M. Yandell, J. Bedell, *BLAST*, O'Reilly, 2003.

[14] S. Kurtz, C. Schleiermacher, REPuter: fast computation of maximal repeats in complete genomes, *Bioinformatics*, 15(8):426–427, 1999.

[15] A. Lefebvre, *Une nouvelle heuristique pour la détection de répétitions sur des génomes complets, pour la comparaison de génomes et pour la compression*, PhD Thesis, Université de Rouen, France, 2003.

[16] A. Lefebvre, T. Lecroq, A heuristic for computing repeats with a factor oracle: Application to biological sequences, *Int. J. Comput. Math.*, 79(12):1303–1315, 2002.

[17] A. Lefebvre, T. Lecroq, Compror: on-line lossless compression with a factor oracle, *Inf. Process. Lett.*, 83(1):1–6, 2002.

[18] A. Lefebvre, T. Lecroq, J. Alexandre, An improved algorithm for finding longest repeats with a modified factor oracle, *Journal of Automata, Languages and Combinatorics*, 8(4):647–658, 2003.

[19] A. Lefebvre, T. Lecroq, H. Dauchel, J. Alexandre, FORRepeats: detects repeats on entire chromosomes and between genomes, *Bioinformatics*, 19(3):319–326, 2003.

[20] A. Mancheron, C. Moan, Combinatorial characterization of the language recognized by factor and suffix oracles, *Proceedings of the Prague Stringology Conference 2004*, to appear.

[21] G. Navarro, M. Raffinot, *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*, Cambridge University Press, 2002.
[22] É. Rivals, M. Dauchet, J-P. Delahaye, O. Delgrange, Fast Discerning Repeats in DNA Sequences with a Compression Algorithm, *Extended abstract in the 8th Workshop on Genome and Informatics (GIW97)*, Tokyo, Japan, 1997.
[23] W.F. Smyth, Repetitive perhaps, but certainly not boring, *Theoret. Comput. Sci.* 249(2):343–355, 2000.
[24] W.F. Smyth, *Computing Patterns in Strings*, Pearson Addison Wesley, 2003.
[25] P. Vincens, L. Buffat, C. André, J.-P. Chevrolat, J.-F. Boisvieux, S. Hazout, A strategy for finding regions of similarity in complete genome sequences, *Bioinformatics*, 14(8):715–725, 1998.
[26] A. C. Yao, The complexity of pattern matching for a random string, *SIAM J. Comput.*, 8(3):368–387, 1979.
[27] J. Ziv, A. Lempel, Compression of individual sequence via variable length coding, *IEEE Transaction on Information Theory*, 24:530–536, 1978.

Thierry Lecroq
ABISS, Faculté des Sciences et des Techniques
Université de Rouen
76821 MONT-SAINT-AIGNAN CEDEX – FRANCE
Email: Thierry.Lecroq@univ-rouen.fr

Arnaud Lefebvre
ABISS, Faculté des Sciences et des Techniques
Université de Rouen
76821 MONT-SAINT-AIGNAN CEDEX – FRANCE
Email: Arnaud.Lefebvre@univ-rouen.fr