Computing repeated factors with a factor oracle

Arnaud Lefebvre *  Thierry Lecroq †

June 8, 2000

Abstract

We present in this article a linear time and space method for the computation of the length of a repeated suffix for each prefix of a given word $p$. Our method is based on the utilization of the factor oracle of $p$ which is a new and very compact structure introduced in [ACR99], used for representing all the factors of $p$. We also exhibit applications where our method really speeds up the computation of repetitions in words including a text data compression scheme.

Keywords: combinatorics on word, string algorithms, repetitions, factor oracle, suffix link, data compression.

1 Introduction

There have been a large number of studies on the problem of finding repetitions (or repeats) in a given word $p$ (see [Smy00]). For instance, the Morris and Pratt [MP70] shift function gives for each prefix of $p$ the length of its longest border. The Boyer-Moore matching shift function (see [BM77], [KMP77] and [Ryt80]) (also called good suffix shift function) gives for each suffix the position of its rightmost recurrence in $p$. We solve here, in linear time and space, the problem of computing for each prefix of $p$ the length of one of its repeated suffix. In most cases, this suffix is closed to the longest repeated suffix. Our main tool is the factor oracle introduced in [ACR99]. This is a new and very compact structure which enables to represent all the factors of $p$. It is more compact and easier to build than the suffix tree [Apo85] of $p$. The factor oracle is an automaton which recognizes at least all the factors of $p$. It has exactly $m+1$ states corresponding to the $m+1$ prefixes of $p$. It has within $m$ and $2m-1$ transitions. Among these transitions at most $m-1$ have to be represented without their labels since for each state (i.e. prefix) all the incoming transitions are labeled by the same letter (i.e. the last letter of the prefix) and can thus be deduced. However it is still

*ESA CNRS 6037-ABISS, Faculté des Sciences, Université de Rouen, 76821 Mont Saint-Aignan Cedex, France. e-mail: arnaud.lefevre@univ-rouen.fr
†LIFAR-ABISS, Faculté des Sciences, Université de Rouen, 76821 Mont Saint-Aignan Cedex, France. e-mail: thierry.lecroq@univ-rouen.fr
an open question to characterize exactly the language recognized by the factor oracle. For our purpose, we use the suffix links used for the construction of the oracle to compute the desired result. Our method is suitable to manipulate very long words such as genomic sequences. We use it to point out, in a very fast manner, regions of interest in such sequences. Using the factor oracle instead of the suffix tree will give us better running times than in [KS99]. Furthermore, we can deal with any kind of alphabet since we do not have to store the transition labels.

This article is organized as follows: section 2 introduces the notion of factor oracle of a word $p$, section 3 shows how to locate a repeated suffix for each prefix of $p$. Section 4 contains our main result which is the computation in linear time and space of the length of a repeated suffix for each prefix of $p$ using the factor oracle of $p$. Section 5 describes some applications of our method for fast computations of repetitions in particular words and finally section 6 gives our conclusions.

2 The factor oracle

First, let us define some notions and definitions. Let $p = p[1..m]$ be a word of length $|p| = m$ over an alphabet $\Sigma$. Let $\varepsilon$ be the empty word ($|\varepsilon| = 0$). A word $w \in \Sigma^*$ is a factor of $p$ if and only if $p$ can be written $p = uuv$ with $u,v \in \Sigma^*$. A word $u \in \Sigma^*$ (resp. $v \in \Sigma^*$) is a prefix (resp. suffix) of $p$ if and only if $p$ can be written $p = uv$ with $v \in \Sigma^*$ (resp. $u \in \Sigma^*$). An occurrence of a factor $w$ of $p$ is denoted by the position $i \in [1..m]$ of its ending letter. A repeated factor of a word $p$ is a factor of $p$ which has at least two distinct occurrences in $p$. Figure 1 illustrates these definitions.

Let $F(p)$ be the set of all the factors of $p$. Let Oracle($p$) be the factor oracle [ACR99] of $p$, which is an automaton that recognizes at least $F(p)$.

**Definition 1** An automaton is a quintuple $(Q, \Sigma, \delta, s, F)$ where:

- $Q$ is a set of states;
- $\Sigma$ is the alphabet;
- $\delta: Q \times \Sigma \rightarrow Q$ is the state transition function;
- $s \in Q$ is the starting state;
- $F \subseteq Q$ is the set of final states.
Figure 2: Factor oracle of $p = aabbab$. All the factors of $aabbab$ are recognized, but some other words are also recognized such $aba$ which is not in $\mathcal{F}(p)$. In this example $\text{REPEAT}_p(6) = ab$.

**Definition 2** Oracle $(p)$, where $p$ is a word of length $m$, is an acyclic automaton, with $Q = \{0, 1, \ldots, m\}$, $s = 0$ and $F = Q$, that recognizes at least the factors of $p$, that has $m + 1$ states and a number of transitions within $m$ and $2m - 1$.

The construction of Oracle $(p)$ is equivalent to the construction of the factor automaton of $p$ (see [Cro86]) without splitting the states. An example of factor oracle is given figure 2.

**Remark.** Every state $i$ is associated with the prefix $p[1..i]$.

In [ACR99], two construction methods of the factor oracle are proposed. We retained the on-line method, which introduces the notions of longest repeated suffix (REPEAT$_p$) and suffix link ($S_p$).

**Definition 3** ([ACR99]) REPEAT$_p(i)$ is the longest repeated suffix of $p[1..i]$.

**Definition 4** ([ACR99]) $S_p[i]$, the suffix link of a state $i$ of Oracle$(p)$, is equal to the state in which REPEAT$_p(i)$ is recognized.

This on-line method builds Oracle $(p)$ by just adding each letter of $p$ one after another from left to right. It means that it is able to build Oracle$(p[1..i+1])$ from Oracle$(p[1..i])$ and $\sigma = p[i+1]$. At the end of the function ADD-LETTER (see [ACR99]), which builds Oracle$(p[1..i+1])$, the suffix link $S_p[i+1]$ of the state $i+1$ is defined. Definitions 3 and 4 insure that $S_p[i+1]$ is equal to the first occurrence of a repeated suffix of $p[1..i+1]$.

**Definition 5** For $0 < i \leq m$, $SP_p(i) = (k_0 = i, \ldots, k_i = 0)$ is the suffix path of state $i$ in Oracle$(p)$, such that for all $r$, $1 \leq r \leq t$, $k_r = S_p[k_{r-1}]$.

During the construction of Oracle$(p[1..i+1])$, two cases are considered:

- the suffix link of the state $i+1$ leads to the starting state 0 ($S_p[i+1] = 0$), thus REPEAT$_p(i+1) = \varepsilon$;
• the suffix link of the state \( i + 1 \) leads to a state \( q \neq 0 \) (\( SP_p[i + 1] = q \neq 0 \)),
thus a suffix of \( \text{Repeat}_p(i + 1) \neq \varepsilon \) ends in position \( SP_p[i + 1] \).

Definition 6 \( \text{Repeat}_p(i + 1) \), for \( 0 \leq i < m \), is equal to \( u \sigma \) where \( u \) is the longest repeated suffix of \( p[1..i] \) ending in position equal to the greatest element of \( SP_p(SP_p[i + 1] - 1) \cap SP_p(i) \), and \( \sigma = p[i + 1] \).

We want to compute \( |\text{Repeat}_p(i)| \) for \( 1 \leq i \leq m \).

We are looking for the length of a common suffix of \( p[1..SP_p[i + 1]] \) and \( p[1..i + 1] \). Looking for this common suffix is equivalent to look for a common suffix of \( p[1..SP_p[i + 1] - 1] \) and \( p[1..i] \), since \( p[i + 1] = p[SP_p[i + 1]] \). The following lemma formalizes this notion.

Lemma 1 \( |\text{Repeat}_p(i + 1)| \), for \( 0 \leq i < m \), is equal to the length of a common suffix of \( p[1..SP_p[i + 1] - 1] \) and \( p[1..i] \), plus \( i \).

Proof. We know that \( p[SP_p[i + 1]] = p[i + 1] = \sigma \). Let \( u \) be a common suffix of \( p[1..SP_p[i + 1]] \) and \( p[1..i + 1] \). Then \( u = v \sigma \), with \( v \in \Sigma^* \). Consequently, \( |\text{Repeat}_p(i + 1)| = |u| = |v| + 1 \), with \( |v| \) equal to the length of a common suffix of \( p[1..SP_p[i + 1] - 1] \) and \( p[1..i] \) (see figure 3).

Figure 3: Illustration of lemma 1.

The problem of finding the length of a common suffix of \( p[1..SP_p[i + 1]] \) and \( p[1..i + 1] \) is equivalent to the search of the length of a common suffix of \( p[1..SP_p[i + 1] - 1] \) and \( p[1..i] \).

The next section shows how to find an occurrence of such a common suffix.

3 Finding an occurrence of a common suffix of two prefixes

We know that the suffix link of a state \( i \) is equal to an occurrence of one of its repeated suffix. The set of all the suffix links can be represented by a tree structure, where the root is the starting state 0 and where for each node \( i \neq 0 \) its parent node is equal to \( SP_p[i] \). Consequently, a common suffix of two prefixes of \( p \) ends in the position given by the greatest common state of their suffix paths.
Figure 4: An occurrence of a common suffix of $p[1..S_p[i + 1] - 1]$ and $p[1..i]$ is equal to $j$.

(lowest common ancestor in the tree, see figure 4). However, we will compute the length of a repeated suffix of each prefix of $p$ in an overall linear time and space complexity without using the notion of lowest common ancestor.

**Definition 7**  
ComSuff$(n_1,n_2)$, the greatest element of $SP_p(n_1) \cap SP_p(n_2)$ is equal to an occurrence of a common suffix of $p[1..n_1]$ and $p[1..n_2]$.

During the construction of Oracle$(p[1..i+1])$ from Oracle$(p[1..i])$ and $p[i+1]$, the backward jumps on the suffix path end when a state $j$ is reached such that $\delta(j,p[i+1])$ is already defined. The fact that $\delta(j,p[i+1])$ is defined insures (by construction of Oracle$(p[1..i+1])$) that there exists a state in $SP_p(S_p[i+1] - 1)$ such that its suffix link is equal to $j$ or $S_p[i+1] - 1 = j$ (lemma 8 in [ACR99]).

**Definition 8**  
$\pi_1$ is equal to the state of $SP_p(i)$ such that $S_p[\pi_1] = j$.

**Definition 9**  
$\pi_2$ is equal to the state of $SP_p(S_p[i+1] - 1)$ such that $S_p[\pi_2] = j$, if $S_p[i+1] - 1 \neq j$, or is equal to $j$ if $S_p[i+1] - 1 = j$.

The following lemma shows that, once $\pi_1$ and $\pi_2$ are found, we know that an occurrence of a common suffix of $p[1..S_p[i+1] - 1]$ and $p[1..i]$ is equal to position $S_p[\pi_1] = \pi_2 = j$ if $S_p[i+1] - 1 = j$, to position $S_p[\pi_1] = S_p[\pi_2] = j$ otherwise. Figure 5 illustrates this situation and an example is given figure 6.

**Lemma 2**  
An occurrence of a common suffix of $p[1..S_p[i + 1] - 1]$ and $p[1..i]$ is equal to position $S_p[\pi_1] = \pi_2 = j$ if $S_p[i + 1] - 1 = j$, to position $S_p[\pi_1] = S_p[\pi_2] = j$ otherwise.
Figure 5: The general situation: dotted arrows from right to left represent a series of suffix links, dashed arrows represent a simple suffix link, plain arrows represent transitions, and dotted arrows from left to right represent series of transitions.

Figure 6: Example over the word $acaadaabdb$ where the greatest common state of $SP_\sigma(S_p[11] - 1) \cap SP_\sigma(10)$ is 0. In this case, $i = 10$, $\sigma = b$, $\pi_1 = 5$, $\pi_2 = 1$. Only direct transitions of the form $\delta(i, p[i]) = i + 1$ and interesting suffix links have been represented.

Proof. By induction on $p[i..S_p[i+1]-1]$ and $p[i..i]$ using definitions 7, 8 and 9.

The next section presents a simple and fast method to compute $|Rep_i|$, for each state $i$ of Oracle($p$).

4 Computing the length of repeated factors of a word

We propose a simple and linear method to compute $|Rep_i|$, for each state $i$ of Oracle($p$).

Definition 10 Let $hrs$ be an array of $m + 1$ integers such that $\forall i, 0 \leq i < m$:

$$hrs[i + 1] = \begin{cases} 0 & \text{if } S_p[i + 1] = 0 \\ hrs[\pi_1] + 1 & \text{if } \pi_2 = S_p[\pi_1] \\ \min(hrs[\pi_1], hrs[\pi_2]) + 1 & \text{otherwise} \end{cases}$$

$hrs[0]$ is set to 0.

We will now prove that the array $hrs$ computes the length of a repeated suffix for each prefix of $p$. 
Figure 7: Particular case where $q - 1 = \pi_2 = S_p[\pi_1]$.

Figure 8: Illustration of the particular case with the word $abcabc$. In this case, $i = 5$, $\sigma = c$ and $q - 1 = \pi_2 = S_p[\pi_1] = 2$.

Lemma 3 $\forall i, 0 \leq i < m, lrs[i+1] = |\text{REP}_p(i+1)|$.

Proof. By recurrence:

- $i = 0$: $S_p[1] = 0$ implies $\text{REP}_p(1) = \text{REPEAT}_p(1) = \varepsilon$. The condition (1) is verified, then $lrs[1] = 0 = |\text{REP}_p(1)|$;
- $i > 0$: assume that $\forall j, 0 \leq j \leq i$, $lrs[j] = |\text{REP}_p(j)|$. At the end of the construction of $\text{Oracle}(p[1..i+1])$, two cases have to be considered:
  - $S_p[i + 1] = 0$: similar to the case $i = 0$;
  - $S_p[i + 1] = q$ ($q \neq 0$), in this situation two cases can arise:
    - $q - 1 = \pi_2 = S_p[\pi_1]$ (see the general situation figure 7, an example is given figure 8): it means that there exists a transition from $q - 1$ to $q$ labeled by $p[i + 1]$. We are in the case (2), then $lrs[i + 1] = lrs[\pi_1] + 1$. According to lemma 2 with $\pi_2 = q - 1$, and lemma 1, $lrs[i + 1] = |\text{REP}_p(i + 1)|$;
    - $q - 1 \neq S_p[\pi_1]$ (see figure 5): we are in the case (3), then $lrs[i + 1] = \min(lrs[\pi_1], lrs[\pi_2]) + 1$. According to lemma 2, a common suffix of $p[1..q - 1]$ and $p[1..i]$ ends in the position $S_p[\pi_1] = S_p[\pi_2] = j$. Its length is the minimum between $lrs[\pi_1]$ and $lrs[\pi_2]$ (see figure 9). Lemma 1 implies $lrs[i + 1] = |\text{REP}_p(i + 1)|$. □

The implementation of this method is straightforward. The suffix link of every new state $i + 1$ ($S_p[i + 1]$) is defined at the end of the function $\text{NEWADDLETTER}$ (see figure 12), which is a new version of the function $\text{ADDLETTER}$.
in [ACR99]. To be computed, \( brs[i + 1] \) needs only \( S_p[i + 1] \) and \( \pi_1 \). These two values are defined during the execution of the function NEWADDLETTER. The value of \( brs[i + 1] \) is computed by the function LENGTHREPEATEDSUFFIX (see figure 13), which takes \( \pi_1 \) and \( S_p[i + 1] \) as arguments. The next proposition states that the value of \( \pi_1 \) is correct.

**Proposition 1** The value of \( \pi_1 \), given as argument to the function LENGTHREPEATEDSUFFIX, respects definition 8.

**Proof.** By definition 8, \( \pi_1 \) must be the state of \( SP_p(i) \) such that \( S_p[\pi_1] = j \). At the beginning of the function NEWADDLETTER, \( \pi_1 \) is set to \( i \) and \( j \) to \( S_p[i] \), thus \( S_p[\pi_1] = j \). During the while loop, \( \pi_1 \) is set to \( j \) and \( j \) to \( S_p[j] \), thus \( S_p[\pi_1] = j \). Then, at the end of the while loop \( S_p[\pi_1] = j \). The value of \( \pi_1 \) given as argument of the function LENGTHREPEATEDSUFFIX respects its definition.

The next lemma proves the correctness of the function LENGTHREPEATEDSUFFIX.

**Lemma 4** The function LENGTHREPEATEDSUFFIX\((\pi_1, S_p[i+1])\) correctly computes \( brs[i+1] \).

**Proof.** The function LENGTHREPEATEDSUFFIX considers two cases:

- \( S_p[i + 1] = 0 \), then \( brs[i + 1] \) is set to 0. The case (1) of the definition 10 is verified;

- \( S_p[i + 1] \neq 0 \), then \( brs[i + 1] \) is set to the value returned by the function LENGTHCOMMONSUFFIX (figure 14), plus 1. Consider now the function LENGTHCOMMONSUFFIX:

  - if \( \pi_2 = S_p[i + 1] - 1 \) is equal to \( S_p[\pi_1] \) (case (2) of definition 10), then the function LENGTHCOMMONSUFFIX returns \( brs[\pi_1] \), LENGTHREPEATEDSUFFIX\((\pi_1, S_p[i+1])\) returns \( brs[i+1] \), and, consequently, the case (2) of definition 10 is verified;
Figure 10: This example over the word \textit{abababdbabc}, where the values of \texttt{brc} are computed (visible on the bottom-right of each state), shows that some values are not the best ones: $S_p[11] = 4$ and $\text{Rep}_p(11) = bc$ then $\texttt{brc}[11] = 2$. The best value for $\texttt{brc}[11]$ should be 3 because $\text{Repeat}_p(11) = abc$ (occurrences 7 and 11). This error is due to the fact that a suffix link is equal to a state in which $\text{Repeat}_p$ is recognized but not always to one of the occurrences of $\text{Repeat}_p$.

Figure 11: \textit{Oracle(ababababab)}. The values of \texttt{brc} are written on the bottom-right of each state, and are equal to the length of $\text{Repeat}_p$ for each state.

- otherwise, if $\pi_2 \neq S_p[i + 1] - 1$, the while loop looks for the state $\pi_2$ of $SP_p(S_p[i + 1] - 1)$ such that $S_p[\pi_2] = S_p[\pi_1] = j$. Then, the function $\text{LengthCommonSuffix}$ returns $\min(\texttt{brc}[\pi_1], \texttt{brc}[\pi_2])$, and the function $\text{LengthRepeatedSuffix}(\pi_1, S_p[i + 1])$ returns $\min(\texttt{brc}[\pi_1], \texttt{brc}[\pi_2]) + 1$, thus the case (3) of definition 10 is verified.

The three cases of definition 10 are verified, consequently $\text{LengthRepeatedSuffix}(\pi_1, S_p[i + 1])$ computes $\texttt{brc}[i + 1]$.

Figures 10 and 11 give two examples of values of the array \texttt{brc}.

The following lemma states the correctness of the function $\text{NewAddLetter}$.

\textbf{Lemma 5} The function $\text{NewAddLetter}$ computes $\textit{Oracle}(p[1..i + 1])$ and $\texttt{brc}[i + 1]$ from $\textit{Oracle}(p[1..i])$, $\sigma = p[i + 1]$ and $\texttt{brc}[0..i]$.

\textbf{Proof}. Using lemma 4 and definition 10.
NEWADDLETTER(Oracle(p[1..i]), σ)
1  Create a new state i + 1
2  δ(i, σ) ← i + 1
3  j ← S_p[i]
4  π_1 ← i
5  while j > -1 and δ(j, σ) is undefined do
6     δ(j, σ) ← i + 1
7     π_1 ← j
8     j ← S_p[j]
9  if j = -1 then
10     s ← 0
11  else  s ← δ(j, σ)
12  S_p[i + 1] ← s
13  b_rσ[i + 1] ← LENGTHREPEATEDSUFFIX(π_1, S_p[i + 1])
14  return Oracle(p[1..i]σ)

Figure 12: Algorithm of the function NEWADDLETTER.

LENGTHREPEATEDSUFFIX(π_1, s)
1  if s = 0 then
2     return 0
3  else  return LENGTHCOMMONSUFFIX(π_1, s - 1) + 1

Figure 13: Finding the length of a repeated suffix of p[1..i + 1].

LENGTHCOMMONSUFFIX(π_1, π_2)
1  if π_2 = S[π_1] then
2     return b_rσ[π_1]
3  else  while S[π_2] ≠ S[π_1] do
4     π_2 ← S[π_2]
5  return min(b_rσ[π_1], b_rσ[π_2])

Figure 14: Finding a common suffix of p[1..i] and p[1..S[i + 1] - 1].
The following theorem establishes the correctness of the computation of the array $brs$.

**Theorem 1** The algorithm $\text{Oracle-on-line}(p[1..m])$ in [ACR99], using the function $\text{NewAddLetter}$, instead of $\text{Add-Letter}$, builds $\text{Oracle}(p)$ and computes $brs[i]$, $\forall i, 0 \leq i \leq m$.

**Proof.** By induction on word $p$ using lemma 5 and theorem 1 in [ACR99].  

We will now give the complexity of the computation of the oracle together with the array $brs$.

**Theorem 2** The complexity of $\text{Oracle-on-line}(p[1..m])$ in [ACR99], using the function $\text{NewAddLetter}$, instead of $\text{Add-Letter}$, is $O(m)$ in time and space.

**Proof.** Theorem 2 in [ACR99] proves that the construction of $\text{Oracle}(p)$ is linear in time and space. The construction of $brs[0..m]$ is linear in space, since each value of this array can be stored in constant space. The only point to verify is that the construction of $brs[0..m]$ is linear in time. The two first cases of definition 10 are solved in constant time. The only problem comes from the third case which implies backward jumps on $SP_p(S_p[i+1] - 1)$. The transition $\delta(j, S_p[i+1])$ has been created during the construction of $\text{Oracle}(p[1..S_p[i+1]])$. Then the number of backward jumps on $SP_p(S_p[i+1] - 1)$ to find $\pi_2$ is less or equal to the number of backward jumps performed during the construction of $\text{Oracle}(p[1..S_p[i+1]])$. The overall number of such backward jumps is linear during the computation of $\text{Oracle}(p)$ (see proof of theorem 2 in [ACR99]). Consequently, the construction of $brs[0..m]$ is linear in time. 

5 Applications and experiments

We are now exhibiting a few applications where the computation of repetitions is of interest. The array $brs$ gives us the lengths of long repeated factors in a string. The suffix links indicate their positions. Consequently, these two structures point out long repetitions in strings. Genomic sequences can be viewed as long strings on a four letter alphabet for DNA and RNA, and a twenty letter alphabet for proteins. Such sequences may be as long as 3 billion letters for the human genome. When dealing with such huge words, we want to get a compact representation of all the factors of the word, this is the case with the factor oracle. The array $brs$ can also help to compute repetitions in a word or compare the repetitions within two words. It also gives a natural data compression scheme.
5.1 Looking for repetitions in genomic sequences: application to chromosomes II and IV of *Arabidopsis thaliana*

Genomic sequences are a very good support to test our method since:

- during the evolution, they go through series of mutations and especially gene duplications and translocations which represent very large repeated blocks;
- some regions contain small but near and numerous repetitions (micro-satellites, ...).

We applied our method to the detection of long repetitions in long genomic sequences and especially on chromosomes II ($\approx 20 \times 10^9$ base pairs (bp)) and IV ($\approx 17 \times 10^9$ bp) of a plant, *Arabidopsis thaliana*: the total running time\(^1\) to find the occurrences of a repeated region is approximately 28 seconds (for each chromosome). After a study of these repetitions, we localized on chromosome IV of *Arabidopsis thaliana*, a 30,318 bp repetition, which is in fact a concatenation of ten identical regions (more than 3,000 bp long each). In GenBank\(^2\), this region corresponds to putative proteins: the problem is that, statistically, this event is not due to a random phenomenon.

Another relevant example has been obtained on this chromosome. Figure 15, taken from [MIPS99] (also available on the MIPS web site\(^3\)), represents approximate repetitions on chromosome IV of the plant *Arabidopsis Thaliana*. According to the heavy method they used (method we tested), it took them a few hours to obtain it. This is due to the fact that they allow errors in the repetitions. With our method, it took us 1 minute to obtain figure 16. Although we look for exact repetitions we can easily see that the general aspect of our result is equivalent to the one of figure 15.

Our method, which is very fast, allows to detect all kind of exact repetitions in genomic sequences and more generally in texts: thus a lot of applications could be developed such as satellites detection, duplicated genes detection, text compression.

5.2 Data compression

Looking for repetitions in texts naturally leads to questions about data compression. A lot of data compression methods have already been developed ([ZL77], [ZL78], [Gal78], [WNC87], [Fen96]). Since our method allows to detect repetitions very fast and in linear space, we developed a data compression method.

Its principle is very simple. Every time that a new letter $(i + 1)$ is added, $S_p(i + 1)$ and $b[a[i + 1]$ are computed. If $S_p(i + 1) = 0$, it means that it is a new letter: it is coded as a single letter. If $S_p(i + 1) \neq 0$, it means that we have

\(^1\)on a PC running Linux with two 500MHz processors and 1 Go RAM.
\(^3\)Munich Information Center for Protein Sequences, http://webvr.mips.biochem.mpg.de
Figure 15: This figure, available on the MIPS web site, represents approximate repetitions on chromosome IV of the plant Arabidopsis thaliana. This results has been obtained after running approximately 12000 BLASTs [AGM+90] between fragments of this chromosome and itself, and after a study of the statistics of these BLASTs. They did not consider the centromer region (gray) which is a particular region on chromosomes.

Figure 16: Exact repetitions of length greater than 75 on chromosome IV of the plant Arabidopsis thaliana, obtained with our method in 1 minute (the centromer region is kept: it does not influence the result.
<table>
<thead>
<tr>
<th>file</th>
<th>gzip</th>
<th></th>
<th></th>
<th></th>
<th>our approach</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>bzip2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ChrIV</td>
<td>27</td>
<td>230</td>
<td>2</td>
<td>43</td>
<td>40</td>
<td>10</td>
<td>% ct dt</td>
<td>% ct dt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ChrII</td>
<td>27</td>
<td>261</td>
<td>2</td>
<td>43</td>
<td>46</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>27</td>
<td>40</td>
</tr>
<tr>
<td>bib</td>
<td>31</td>
<td>0.14</td>
<td>0.01</td>
<td>43</td>
<td>0.18</td>
<td>0.08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>24</td>
<td>0.17</td>
</tr>
<tr>
<td>book1</td>
<td>40</td>
<td>1.18</td>
<td>0.08</td>
<td>56</td>
<td>2.08</td>
<td>0.55</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>30</td>
<td>1.47</td>
</tr>
<tr>
<td>book2</td>
<td>33</td>
<td>0.68</td>
<td>0.06</td>
<td>48</td>
<td>1.32</td>
<td>0.04</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>25</td>
<td>1.1</td>
</tr>
<tr>
<td>progc</td>
<td>33</td>
<td>0.04</td>
<td>0.01</td>
<td>52</td>
<td>0.07</td>
<td>0.04</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>31</td>
<td>0.07</td>
</tr>
<tr>
<td>trans</td>
<td>20</td>
<td>0.08</td>
<td>0.01</td>
<td>52</td>
<td>0.07</td>
<td>0.04</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>19</td>
<td>0.14</td>
</tr>
</tbody>
</table>

Figure 17: Data compression results: our method has been compared to gzip and bzip2. Files ChrII and ChrIV contain the DNA sequences of chromosomes II and IV of Arabidopsis thaliana. Files bib, book1, book2, progc and trans are taken from the Calgary Corpus. For each method we give the compression ratio (%), coding time (ct) and decoding time (dt) are then given in seconds.

found a repeated suffix of length $br[i + 1]$ and one of its occurrences is equal to $S_p(i + 1)$. While $br[i + 1]$ is greater or equal to the difference between $i + 1$ and the last position in which a compression has been performed, there is no need to code it. When $br[i + 1]$ is less than this difference, we code the previous repetition as a couple (length, position).

Some experimental results are presented in figure 17. These first results have been obtained without optimization.

5.3 Pairwise comparison

A part of the study of genomic sequences is the search of similarities between sequences. Another possible application of our approach is to perform pairwise comparisons between genomic sequences.

Consider two sequences $s_1$ and $s_2$ over an alphabet $\Sigma$, and a letter $\eta \notin \Sigma$. The computation of Oracle($p = s_1\eta s_2$) enables the detection of a common factor of $s_1$ and $s_2$ in linear time and space.

6 Conclusion and open problems

We presented a linear time and space complexity computation of a repeated suffix for each prefix of a given word $p$, using the factor oracle of $p$. This suffix is long enough for a lot of applications. The factor oracle is used since it is a very compact structure for representing all the factors of $p$. Our method enables us to locate efficiently exact repeats in $p$. Our goal is to extend the method to deal with approximate repeats and to the computation the longest exact repeats in linear time. An open problem is to find the longest repeated suffix of each prefix of $p$ in linear time and space with the factor oracle.
Acknowledgements

We are grateful to Mathieu Raffinot for his help on understanding the subtleties of the Oracle, to Joël Alexandre for useful comments on the biological applications of our method and to Jean-Pierre Duval for improving earlier versions of this article.

References


