

Drastic improvements over repeats found with a factor oracle

Arnaud Lefebvre^{1*}, Thierry Lecroq², and Joël Alexandre¹

¹ ABISS-UMR CNRS 6037, Faculté des Sciences, Université de Rouen,
76821 Mont-Saint-Aignan Cedex, France.

{arnaud.lefebvre,joel.alexandre}@univ-rouen.fr

² LIFAR-ABISS, Faculté des Sciences, Université de Rouen,
76821 Mont-Saint-Aignan Cedex, France.

thierry.lecroq@univ-rouen.fr

Abstract. We first give some experimental evidences of the error rate on the length of the repeats of a string p found using the factor oracle of p . We show then how to improve the length of the repeats. Examples of improvements are given for finding repeats in genomic sequences and using repeats for data compression.

1 Introduction

Finding repeats in strings is of great interest in areas such as bioinformatics and data compression. There exist exhaustive methods to find all the repeats in a string (see [2] and [4]). The new challenge consists in dealing with huge strings such as those generated in computational biology. In [5] we introduced an on-line linear heuristic method to compute repeats in a string p using the factor oracle of p . We also showed that this method is very useful when applied on genomic sequences. However this method is a heuristic and we were not able to precisely characterize the rate of the approximation. We give here some empirical evidences of this rate. We then show how to improve the length of the repeats found using the factor oracle this leads to introduce a data structure called the *repeat oracle*. Experiments on genomic sequences and in data compression show that the new method drastically improves the previous one.

The remaining of this article is organized as follows. The next section recalls some notations and background notions on strings and factor oracles. Section 3 presents experimental estimations on the error rate of the length of the repeats found using the factor oracle. In Sect. 4 we introduce the improvement and in Sect. 5 we show examples on genomic sequences and in data compression. Finally, in Sect. 6 we present our conclusions.

* The work of the two first authors was partially supported by a NATO grant PST.CLG.977017.

2 Notations and background notations

Let $p = p[1..m]$ be a word of length $|p| = m$ over an alphabet Σ . An *occurrence* of a factor w of p is denoted by the position $i \in \{1, \dots, m\}$ 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 .

The factor oracle of a word p of length m is a deterministic finite automaton (Q, q_0, F, δ) where $Q = \{0, 1, \dots, m\}$ is the set of states, $q_0 = 0$ is the starting state, $F = Q$ is the set of terminal states and δ is the transition function. The factor oracle of a word p 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 p , including the empty one); it has within m and $2m - 1$ transitions; it recognizes at least all the factors of p . It can be built on-line in linear time (see [1]). There is a bijection between the states and the length of the $m + 1$ prefixes of p (including the empty one). Each transition leading to state i is labeled by $p[i]$. We distinguish two kinds of transitions: transitions from state i to state $i + 1$ are called *internal transitions* and transitions from 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 word p and at most $m - 1$ external transitions without their label. All the other information can be deduced from the word p . This representation is completely independent from the underlying alphabet. The factor oracle of p recognizes at least all the factors of p and slightly more words. The exact characterization of the language recognized by the factor oracle is still an open question. An example of factor oracle is given Fig. 1.

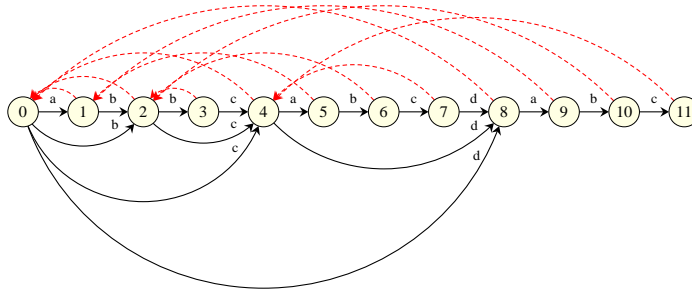


Fig. 1. Factor oracle of $p = abbcabcdabc$. Dash arrows represent the suffix links. All transitions leading to state i are labeled by $p[i]$. There is always a transition from state $i - 1$ to state i . Thus this factor oracle can be represented by p itself and the list $((0, 2), (0, 4), (0, 8), (2, 4), (4, 8))$ of the external transitions without their label. All the states are terminal. All the factors of p are recognized. Some more words that are not factors but subwords of p , such as $abca$, are also recognized.

We denote by $LRS(p)$ the longest repeated suffix of p : $LRS(p) = \max\{v \mid v \text{ is a suffix of } p \text{ and } v \text{ is a factor of } p[1..|p| - 1]\}$.

We recall the definition of the suffix link for a state i of the factor oracle of a word p .

Definition 1 ([1]) $S[i]$, the suffix link of a state i of the factor oracle of a word p , is equal to the state in which the longest repeated suffix of $p[1..i]$ is recognized: $S[i] = \delta(0, LRS(p[1..i]))$.

The state $S[i]$ is equal to an occurrence of a repeated suffix of $p[1..i]$.

Figure 2 presents the pseudo-code for the computation of the factor oracle of a word p of length m .

```

ORACLE( $p, m$ )
1 Create state 0
2  $S[0] \leftarrow -1$ 
3 for  $i \leftarrow 1$  to  $m$ 
4     do ADDLETTER( $i$ )
ADDLETTER( $i$ )
1 Create state  $i$ 
2  $\delta(i - 1, p[i]) \leftarrow i$ 
3  $k \leftarrow S[i - 1]$ 
4 while  $k > -1$  and  $\delta(k, p[i])$  is undefined
5     do  $\delta(k, p[i]) \leftarrow i$ 
6      $k \leftarrow S[k]$ 
7 if  $k = -1$ 
8     then  $S[i] \leftarrow 0$ 
9     else  $S[i] \leftarrow \delta(k, p[i])$ 
    
```

Fig. 2. The algorithm ORACLE(p, m) builds the factor oracle of the word p of length m . The function ADDLETTER(i) builds the factor oracle of $p[1..i]$ from the factor oracle of $p[1..i - 1]$.

3 Computing repeats with a factor oracle

In [5] we describe a method to compute, for each prefix $p[1..i]$ of p , the length of one of its repeated suffixes, such that $S[i]$ is one of its occurrences. This length is denoted by $lrs[i]$.

We proved in [5] that $lrs[i]$ is a good approximation of $LRS(p[1..i])$ and $lrs[i] \leq |LRS(p[1..i])|$ always holds.

Figure 3 shows the pseudo-code for the computation of the factor oracle of a word p of length m together with the array lrs .

Since it is a heuristic, we performed some experiments in order to estimate the number of states i where $lrs[i]$ is different from $|LRS(p[1..i])|$. The exact values were computed using a classical dotplot.

```

ORACLEANDLRS( $p, m$ )
1 Create state 0
2  $S[0] \leftarrow -1$ 
3 for  $i \leftarrow 1$  to  $m$ 
4   do NEWADDLETTER( $i$ )
NEWADDLETTER( $i$ )
1 Create state  $i$ 
2  $\delta(i-1, p[i]) \leftarrow i$ 
3  $k \leftarrow S[i-1]$ 
4  $\Pi_1 \leftarrow i-1$ 
5 while  $k > -1$  and  $\delta(k, p[i])$  is undefined
6   do  $\delta(k, p[i]) \leftarrow i$ 
7      $\Pi_1 \leftarrow k$ 
8      $k \leftarrow S[k]$ 
9 if  $k = -1$ 
10  then  $S[i] \leftarrow 0$ 
11     $lrs[i] \leftarrow 0$ 
12  else  $S[i] \leftarrow \delta(k, p[i])$ 
13     $lrs[i] \leftarrow \text{LENGTHCOMMONSUFFIX}(\Pi_1, S[i] - 1) + 1$ 
LENGTHCOMMONSUFFIX( $\Pi_1, \Pi_2$ )
1 if  $\Pi_2 = S[\Pi_1]$ 
2   then return  $lrs[\Pi_1]$ 
3   else while  $S[\Pi_2] \neq S[\Pi_1]$ 
4     do  $\Pi_2 \leftarrow S[\Pi_2]$ 
5   return  $\min(lrs[\Pi_1], lrs[\Pi_2])$ 

```

Fig. 3. The algorithm $\text{ORACLEANDLRS}(p, m)$ builds the factor oracle of the word p of length m together with the array lrs . The function $\text{NEWADDLETTER}(i)$ computes the factor oracle of $p[1..i]$ and the value $lrs[i]$ from the factor oracle of $p[1..i-1]$ and $lrs[1..i-1]$. The function $\text{LENGTHCOMMONSUFFIX}(\Pi_1, \Pi_2)$ finds the length of a common suffix ending at the position Π_1 and Π_2 by traversing the suffix links.

Figures 4 and 5 show results obtained on the chromosome I of the model plant *Arabidopsis thaliana*. It contains a little more than 31 millions of symbols. As far as the dotplot method is very slow, we performed this comparison on a sliding window. In order to show that the window size does not constitute a bias in our measure we give the results for two window sizes: 25,000 and 100,000.

The percentage of errors is about 40% for the two window sizes. This figure seems quite large, however, the average difference between the lrs values and the dotplot values is less than one.

Experiments were conducted on a large number of sequences and were all consistent with those presented above. It means that we are wrong in a lot of cases but we are not so far from the exact values one could compute with an exact method and we are much more faster.

In the next section we describe an experimental improvement of this method which reduces significantly the number of errors and the average error.

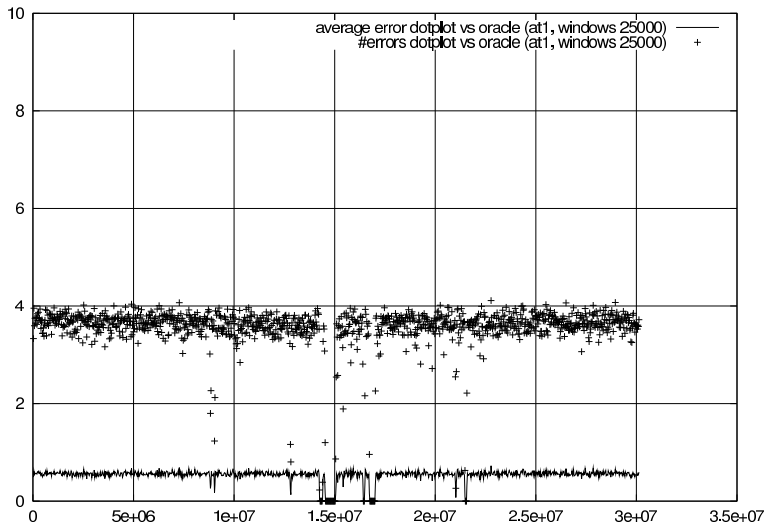


Fig. 4. Results obtained with a factor oracle on the chromosome I of the model plant *Arabidopsis thaliana*. The size of the windows is 25,000 symbols. Values on the X-axis are the starting positions of the windows on the chromosome. On the Y-axis the number of errors represents the percentage of errors divided by ten ; the average error represents the average difference between *lrs* values and the dotplot values. The number of errors for each window is about 40% and the average error is a bit less than one.

4 A better heuristic

Since the suffix link of each state i leads to a state j smaller or equal to the first occurrence of the longest repeated suffix of $p[1..i]$, an occurrence of $p[i - lrs[i]..i]$ can be equal to a state k such that $j < k < i$ and $S[k] = S[i] = j$.

Figure 6 illustrates this situation.

Thus the idea of the improvement is, every time that a *lrs* value is computed, to verify if there exists such another occurrence of a longer repeated suffix. If it is the case, then the suffix link is updated to the new occurrence and the *lrs* value is increased by one.

Figure 7 presents the pseudo-code of the improved method.

This new structure is then called *repeat oracle*. It is even more difficult to characterize than the factor oracle. However it is much more accurate regarding the detection of repeats.

5 Applications

We now show two applications where the improved method reveals very useful.

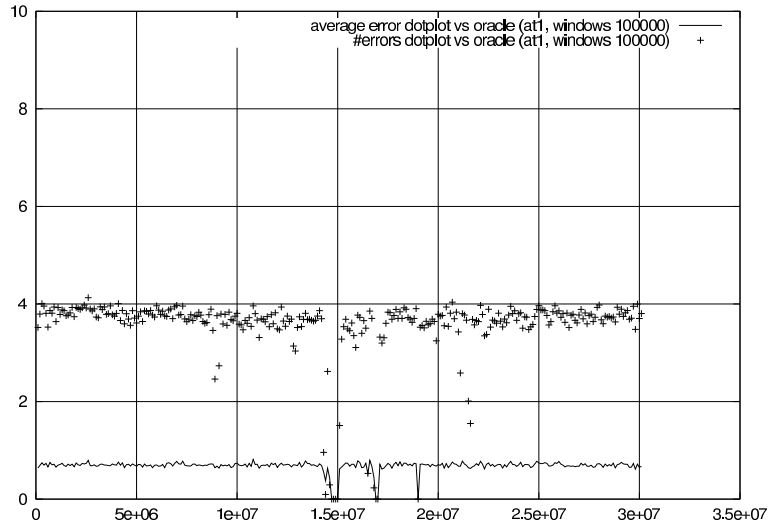


Fig. 5. Results obtained with a factor oracle on the chromosome I of the model plant *Arabidopsis thaliana*. The size of the windows is 100,000 symbols. Values on the X-axis are the starting positions of the windows on the chromosome. On the Y-axis the number of errors represents the percentage of errors divided by ten ; the average error represents the average difference between *lrs* values and the dotplot values. The number of errors for each window is about 40% and the average error is a bit less than one.

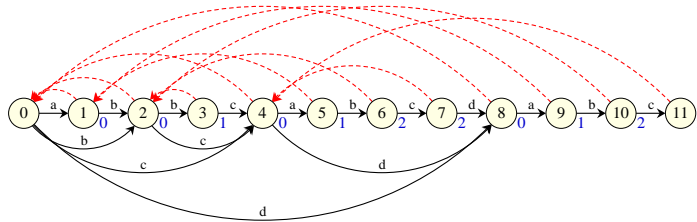


Fig. 6. Factor oracle of *abcbabcdabc*. Dash arrows represent the suffix links. Values near the states are the *lrs* values. The detected repeated suffix of *abcbabcdabc* is *bc* (length 2). However, there exists an occurrence of *abc* (length 3) in position 7 which is a longer repeated suffix than *bc*. The improvement of this method will move the suffix link to state 7, and increase the *lrs* of state 11 by one.

```

REPEATORACLE( $p, m$ )
1 Create state 0
2  $S'[0] \leftarrow -1$ 
3 for  $i \leftarrow 1$  to  $m$ 
4     do NEWIMPROVEDADDLETTER( $i$ )
NEWIMPROVEDADDLETTER( $i$ )
1 Create state  $i$ 
2  $\delta(i-1, p[i]) \leftarrow i$ 
3  $k \leftarrow S'[i-1]$ 
4  $\Pi_1 \leftarrow i-1$ 
5 while  $k > -1$  and  $\delta(k, p[i])$  is undefined
6     do  $\delta(k, p[i]) \leftarrow i$ 
7          $\Pi_1 \leftarrow k$ 
8          $k \leftarrow S'[k]$ 
9 if  $k = -1$ 
10 then  $S'[i] \leftarrow 0$ 
11      $lrs[i] \leftarrow 0$ 
12 else  $S'[i] \leftarrow \delta(k, p[i])$ 
13      $lrs[i] \leftarrow \text{NEWLENGTHCOMMONSUFFIX}(\Pi_1, S'[i] - 1) + 1$ 
14  $k \leftarrow \text{FINDBETTER}(i, p[i - lrs[i]])$ 
15 if  $k \neq 0$ 
16 then  $lrs[i] \leftarrow lrs[i] + 1$ 
17      $S'[i] \leftarrow k$ 
18  $T[S'[i]] \leftarrow T[S'[i]] \cup \{i\}$ 
FINDBETTER( $i, a$ )
1 for all the elements  $j$  of  $T[i]$  in increasing order
2     do if  $lrs[j] = lrs[i]$  and  $p[e - lrs[i]] = a$ 
3         then return  $j$ 
4 return 0
NEWLENGTHCOMMONSUFFIX( $\Pi_1, \Pi_2$ )
1 if  $\Pi_2 = S'[\Pi_1]$ 
2     then return  $lrs[\Pi_1]$ 
3 else while  $S'[\Pi_2] \neq S'[\Pi_1]$ 
4     do  $\Pi_2 \leftarrow S'[\Pi_2]$ 
5 return  $\min(lrs[\Pi_1], lrs[\Pi_2])$ 

```

Fig. 7.

5.1 Repeat detections

In order to estimate the improvement, the same experiments as those presented in the previous section have been performed. Results are shown figure 8 only for window size 100,000.

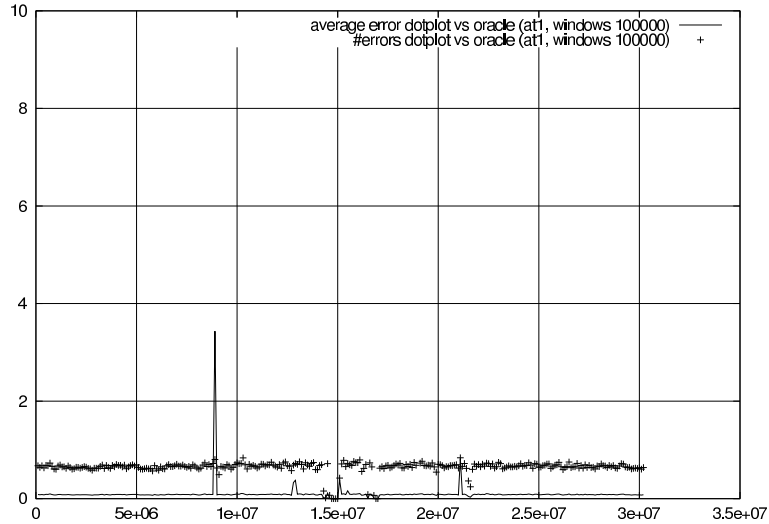


Fig. 8. Results obtained with a repeat oracle on the chromosome I of the model plant *Arabidopsis thaliana*. The size of the windows is 100,000 symbols. Values on the X-axis are the starting positions of the windows on the chromosome. On the Y-axis the number of errors represents the percentage of errors divided by ten ; the average error represents the average difference between *lrs* values and the dotplot values.

With the improvement, the average number of errors drops to 6% (instead of 40%), and the average error drops to 0.1 (instead of 1). Experiments were conducted on a large number of sequences and were all consistent with those obtained with chromosome I of *Arabidopsis thaliana*. Regarding to these results, this improvement reduces significantly errors generated by the previous method.

In terms of running time, this improvement is slower than the original method. Figure 9 shows that the improved method is twice slower than the original one. This would mean that the improved method is still linear.

5.2 Lossless on-line data compression

Computing the length of long repeated suffix for each prefix of a word p naturally leads to a factorization of p which can be used to compress it [6]. All the first occurrences of the letters of the alphabet in p will be encoded as single letters. All the repeated factors will be encoded as pairs (*length, position*). The encoding is performed simultaneously with the construction of the factor oracle.

	size	original	improved
at1	31	50	85
at4	17.5	28	54
pic	0.5	1	2
book1	0.7	1.5	3
book2	0.6	1.4	3

Fig. 9. Execution times, given in seconds, of the original method and the improved method applied on files containing DNA sequences (at1 and at4 are respectively chromosomes I and IV of *Arabidopsis thaliana*), pictures (pic) or English texts (book1 and book2). Sizes of the files are given in Mb in the first column.

Assume that the factor oracle of $p[1..j]$ has already been built and thus that the prefix $p[1..j]$ of length j of the word p has already been encoded. To encode the suffix $p[j+1..m]$, we then need to find the smallest position $i+1$ strictly greater than j such that $lrs[i+1] < i+1-j$. If $i+1 = j+1$ (or equivalently $lrs[j+1] = 0$) then it means that the letter $p[j+1]$ never occurred in $p[1..j]$ and it will be encoded as a single letter. Otherwise we will represent $p[j+1..i]$ as the pair $(i-j, S[i]-i+j+1)$ since $p[j+1..i] = p[S[i]-i+j+1..S[i]]$. At that time the prefix $p[1..i]$ has been encoded, it remains to encode by the same process the suffix $p[i+1..m]$. The reader can refer to [6] for the encoding details. This compressing method is called `compror`.

The word *aabbabbabbab* which oracle and *lrs* values are given figure 1 will be encoded by $a(1,1)b(1,3)(8,2)$.

The decoding process is straightforward. Given $a(1,1)b(1,3)(8,2)$ it is obvious to retrieve the word *aabbabbabbab*.

As far as it is possible to detect longer repeats with the improved method than with the original one, it has been introduced in `compror`. This new compressing method has been compared to `bzip2` which is based on the Burrows-Wheeler transform [3] and `gzip` based on Ziv-Lempel method [7]. New compression results are given figure 10.

	size	bzip2	gzip	original	improved
at4	17.5	4.7	4.9	5.7	4.8
pic	0.51	0.05	0.05	0.12	0.073
book1	0.77	0.23	0.31	0.43	0.31
book2	0.61	1.5	0.20	0.29	0.23

Fig. 10. Compression results on chromosome IV of *Arabidopsis thaliana*, a picture (pic) and English texts (book1 and book2). The size column gives the size of each file (in Mb), other columns give the sizes of the files after compression with `bzip2`, `gzip`, original `compror` and the improved version of `compror` respectively.

The improved version of `compror` is much better than the original version and beats `gzip` in a large number of cases.

6 Conclusion

We present in this article, for the first time, experimental results that enable to measure the approximation rate of the length of the repeats found on-line and in linear time with a factor oracle. Though the number of errors is quite high, the error is itself closed to one on the average, even for large sequences. Furthermore we showed an improvement of the original method which divides the number of errors by seven and the error itself by ten. This leads to improved results when looking for exact repeats in genomic sequences and when applied to data compression. Furthermore the experimental results indicate that the method, which is still on-line, remains linear. We leave, as an open problem, the exact characterization of the new data structure named *repeat oracle*.

References

1. C. Allauzen, M. Crochemore and 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*, number 1725, pages 291–306, Milovy, Czech Republic, 1999.
2. M. Crochemore and W. Rytter, *Text algorithms*, Oxford University Press, 1994.
3. P. Fenwick, The Burrows-Wheeler transform for block sorting text compression – principles and improvements, *The Computer Journal*, 39(9):731–740, 1996.
4. D. Gusfield, *Algorithms on strings, trees and sequences: computer science and computational biology*, Cambridge University Press, 1997.
5. A. Lefebvre and T. Lecroq, Computing repeated factors with a factor oracle, In L. Brankovic and J. Ryan, editors, *Proceedings of the 11th Australasian Workshop On Combinatorial Algorithms*, pages 145–158, Hunter Valley, Australia, 2000.
6. A. Lefebvre and T. Lecroq, Compror: on-line lossless compression with a factor oracle *Inf. Process. Lett.*, 2002, to appear.
7. J. Ziv and A. Lempel, Compression of individual sequence via variable length coding. *IEEE Transaction on Information Theory*, 24:530–536, 1978.