

Maximal Repetitions and Application to DNA sequences

Mathieu Giraud, Gregory Kucherov

▶ **To cite this version:**

Mathieu Giraud, Gregory Kucherov. Maximal Repetitions and Application to DNA sequences. Journées Ouvertes : Biologie, Informatique et Mathématiques - JOBIM'2000, 2000, Montpellier/France, pp.165–172. inria-00107854

HAL Id: inria-00107854
https://inria.hal.science/inria-00107854

Submitted on 19 Oct 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Maximal repetitions and Application to DNA sequences

Mathieu Giraud [*]
École Normale Supérieure de Lyon, France

Gregory Kucherov [†]
LORIA, Nancy, France

Abstract

In this paper we describe an implementation of Main-Kolpakov-Kucherov algorithm [9] of linear-time search for maximal repetitions in sequences. We first present a theoretical background and sketch main components of the method. We also discuss how the method can be generalized to finding approximate repetitions. Then we discuss implementation decisions and present test examples of running the programs on real DNA data.

Keywords : maximal repetitions, DNA repeats.

Introduction

Successive repetitions of a fragment in DNA sequences often bear important information and is characteristic for many genomic structures (telomer regions, tandem repeats and others). From practical viewpoint, satellites and alu-repeats are involved in chromosome analysis and genotyping, and thus are of great interest to genomic researchers. Tools for finding successive repeats are nowadays an obligatory part of integrated systems for analyzing and annotating whole genomes.

In this paper we describe an implementation of an efficient (linear-time) algorithm for finding so-called maximal repetitions. The algorithm was recently proposed by R.Kolpakov and G.Kucherov [9] and is a modification of Main's algorithm [12].

The paper is organized as follows. In Section 1, we introduce necessary definitions and describe briefly the components of the algorithm, in particular Main and Lorentz's extension functions and s-factorization. In the concluding subsection, we sketch how the algorithm can be modified in order to find *approximate* maximal repetitions, that is maximal repetitions allowing errors of type substitution. In Section 2, we describe the implementation of the algorithm. We justify some implementation decisions, such as computing s-factorization using the DAWG structure. We present also some characteristics and examples of the program, as well as a small graphical interface for visualizing its results. Finally, in Section 3, we describe applications of the program to real DNA sequences.

[*] magiraud@ens-lyon.fr
[†] kucherov@loria.fr

1. Background: Finding all maximal repetitions in linear time

1.1 Notations and problem statement

Consider a word $w = a_1 \dots a_n$ on a finite alphabet $\mathcal{A}$. An integer p is said to be a **period** of w iff $\forall i, 1 \leq i, i+p \leq n, a_i = a_{i+p}$. Each word w has a minimal period that we denote $p(w)$ and call simply **the** period of w. The ratio $e(w) = \frac{|w|}{p(w)}$ is called the **exponent** of w. w is a **repetition** iff $e(w) \geq 2$. In this paper we are interested in subwords of a word which are repetitions. Such a repetition is called **maximal** if it cannot be extended to the right/left by one letter without increasing its period.

Example : In abaacacaba, acac is a repetition which is not maximal, whereas acaca is a maximal repetition.

For the sake of shortness, maximal repetitions will often be simply called **repetitions** in the sequel. Main showed in 1989 that all *leftmost* repetitions can be found in linear time [12]. The goal of this section is to briefly explain the algorithm of [9] which computes *all* repetitions of a given word in linear time.

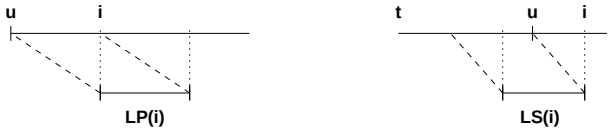
1.2 Main and Lorentz's extension functions [13]

We start with an auxiliary problem. Let $v = tu$ be a word formed by concatenation of two words t and u, with $|t| = m$ and $|u| = n$. We are looking for maximal repetitions which begin in t and end in u.

We need two auxiliary functions:

- $LP(i) = \max\{j | u[1 \dots j] = u[i \dots i+j-1]\}$ for $i \in [2 \dots n]$ and $LP(n+1) = 0$.
- $LS(i) = \max\{j | t[m-j+1 \dots m] = v[m+i-j+1 \dots m+i]\}$ for $i \in [1 \dots n]$.

Informally, $LP(i)$ is the length of the longest prefix of u which is also a prefix of $u[1 \dots n]$, and $LS(i)$ is the length of the longest suffix of t which is also a suffix of $tu[1 \dots i]$.



Computation of LP and LS can be done in $O(n)$ time using an adaptation of the Knuth-Morris-Pratt algorithm [8], explicitly described in [13]. Slightly different techniques of computing these functions are described in [5] and [7].

Now, we can find maximal repetitions with the following theorem:

Theorem 1 (Main and Lorentz) *For $1 \leq j \leq n$, there exists a maximal repetition of period j in $v = tu$ which starts in t, ends in u and has a full period in u iff $LS(j) + LP(j+1) \geq j$. If the inequality holds, then this maximal repetition is $v[|u| - LS(j) + 1 \dots |u| + j + LP(j+1)]$.*

Repetitions which have a full period in t can be found similarly.

1.3 s-factorization and leftmost repetitions

Here we show how Main's algorithm [12] finds the *leftmost* occurrences of all distinct repetitions. We use the *s-factorization* of a word w is $w = u_1 u_2 \dots u_k$, where the *s-factors* u_i's are inductively defined by:

- u_i is the first letter after $u_1 u_2 \dots u_{i-1}$ if this letter does not occur in $u_1 u_2 \dots u_{i-1}$,
- otherwise, u_i is the longest subword after $u_1 u_2 \dots u_{i-1}$ which has at least two occurrences – possibly overlapping – in $u_1 u_2 \dots u_{i-1} u_i$.

Example : The s-factorization of the word acaacacaacaac is a c a aca caaca ac.

The s-factorization (also called Lempel-Ziv factorization in [7], because of its closeness with the factorization used in the famous compression algorithm) was introduced by Crochemore in [3].

It turns out to be a useful tool in finding repetitions, in particular it was used by Main [12] for computing all *leftmost* occurrences of maximal repetitions by the following theorem :

Theorem 2 (Main) *If the s-factorization of w is $u_1 u_2 \dots u_k$, then every leftmost repetition occurrence which ends in u_i has less than $|u_i| + 2|u_{i-1}|$ characters before u_i.*

This theorem implies that if we have the s-factorization, all leftmost occurrences of distinct repetitions can be found in *linear* time by the following algorithm [12] :

Algorithm 1 (Main)
input: `s-factorization u(1)...u(k) of a word w`
output: `all leftmost repetitions of w`

```
For each s-factor u(i) with i >= 2:
 - Compute l = |u(i)| + 2 |u(i-1)|
 - Let t be the suffix of u(1)..u(i-1) of length l,
 - Search, using Main and Lorentz's extension functions, the
   repetitions starting in t and ending in u(i).
```

1.4 Number of repetitions

Kolpakov and Kucherov recently proved the following theorem [10] :

Theorem 3 (Kolpakov and Kucherov) *The number of maximal repetitions in a word is linear in the word length.*

The proof [10] is quite technical. In fact, the exponent sum of such repetitions is also linear, as showed in [9].

1.5 Putting all together : the Main-Kolpakov-Kucherov (MKK) algorithm

In [9], Kolpakov and Kucherov proposed an extension of Main's algorithm to find *all* maximal repetitions.

Algorithm 2 (MKK algorithm)
input: `a word w`
output: `all repetitions of w`

```
1) [Section 2.1] Compute the s-factorization of the input word.

2) [12] Find all repetitions according to Algorithm 1.
        This guarantees finding all leftmost occurrences.

3) [9] For each s-factor, find repetitions occurring inside the factor.
       Each of those repetitions is a copy of another one,
       found earlier.
```

Computing s-factorization takes a linear time (see below) and the two other steps are in $O(n)$ by Theorems 1, 2, and 3. Therefore the whole MKK algorithm takes **linear time**.

1.6 A parenthesis : inexact maximal repetitions

The main default of our previous algorithm, when applied to biology, is that we compute only *exact* repetitions. Repetitions occurring in biological sequences are often *inexact*, therefore we need to define and to compute them.

As noted in the introduction of [15], previous results on repetitions in biosequences concern either exact repetitions, or repeats involving only two elements (*squares*), as in [11]. Sagot and Myers, in [15], found an algorithm in $O(n\mathcal{N})$ time, where $\mathcal{N}$ is a size of a *neighborhood* which can be, at worse, in $O(n)$.

The ρ-value and κ-mismatch repetitions

Here we define a ρ-value which can be a useful tool in quantifying how many errors an *approximate* repetition can count.

We consider only *substitution* errors, and do not consider deletions and insertions. If w_a and w_b are words on $\mathcal{A}$ having the same length, we denote $\sigma(w_a, w_b)$ the classical substitution distance (the *Hamming distance*) between the two words, *ie* the number of character differences between them. Let $w = a_1 \dots a_n$ be a word. If $1 \leq j \leq \frac{|w|}{2}$, we define the ρ-value ρ_j for period j by :

$$\rho_j(v_1 \dots v_n) = \sigma(v_1 \dots v_{n-j}, v_{j+1} \dots v_n) = \#\{i | i \in [1, n-j] | v_i \neq v_{i+j}\}$$

The ρ-value of w is then :

$$\rho(w) = \min\{\rho_j(w) | 1 \leq j \leq \frac{|w|}{2}\}$$

Example : $\rho(\mathsf{ABCABCABCA}) = \rho_3(\mathsf{ABC\ ABC\ ABC\ A}) = 0$
and $\rho(\mathsf{ABXABCABCF}) = \rho_3(\mathsf{AB}X\ \mathsf{ABC}\ \ \mathbf{A}\mathsf{BC}\ \mathbf{F}) = 2$.

Let κ be a positive integer named the *error threshold*. If $\rho_j(w) \leq \kappa$ we say that w has a κ-**mismatch period** j and is a κ-**mismatch repetition**. We denote $J_\kappa(w)$ the set of those periods. So we have :

$$w \text{ is a } \kappa\text{-mismatch repetition } \iff \rho(w) \leq \kappa \iff J_\kappa(w) \neq \emptyset$$

A κ-mismatch repetition is defined as **maximal** if it cannot be extended while preserving the same period. For short, such a repetition is called an **approximate repetition**.

Remark that, by our definition, the repeated pattern can be significantly changed between its first and last occurrences (like ABC ABC ABC FFF FFF FFF which is an 3-mismatch repetition). Therefore, if we are interested in the repetition of a *pattern*, our definition does not fit, except if we assume that $\rho(w) \ll j$ for some $j \in J_\kappa(w)$. But if we want to count the internal repetitions of a word, our ρ-value is good, and we need only to assume $\rho(w) \ll |w|$.

Generalizing Main and Lorentz's extension functions and algorithms

This definition allows us to extend Main and Lorentz's functions LP and LS: given an error threshold κ, extensions LS_κ and LP_κ can be naturally defined. It is then possible to find an analog of Main's theorem and apply a divide-and-conquer algorithm. So we can obtain an algorithm which finds *all* approximate repetitions in $O(\kappa n \log n)$ time. Improving this bound is an interesting problem which is a subject of our current investigation.

2. Implementation of MKK algorithm

2.1 Off-line construction of s-factorization using DAWG

Computing s-factorization can be done in $O(n)$ time. Methods proposed in the literature [5, 14] consist of constructing the s-factorization along with constructing the suffix tree. We adopt another strategy, and construct the s-factorization using the **DAWG** (Directed Acyclic Word Graph), instead of the suffix tree. DAWG is a powerful data structure obtained by identifying isomorphic branches of the uncompacted suffix tree [1, 4]. Besides, we construct the s-factorization in off-line fashion, that is *after* the DAWG is constructed.

A very simple algorithm allows us to obtain the s-factorization *after* the DAWG is constructed. The set of w subwords is denoted $sub(w)$, and if $v \in sub(w)$, $endpos(v)$ is the set of end-positions of v in w. We define on DAWG vertices the function $\tau([u]_w) = \min(endpos(u))$.

Using basic considerations, we can prove that

Lemma 1 *(i) For $u \in sub(w)$ with $u \neq w$, we have $\tau([u]_w) = \min_{\alpha \in \mathcal{A}}(\tau([u\alpha]_w)) - 1$*

(ii) A subword $v = u_i \dots u_j$ of w has an earlier occurrence starting at a position $l < i$ iff $j > \tau([v]_w)$.

Using *(i)*, we label the DAWG vertices with the τ function by a bottom-up traversal taking $O(n)$ time. To compute the s-factorization, we use *(ii)* which implies only a linear run through the word, where we look at the τ function.

Therefore computing the s-factorization from the DAWG can be done in $O(n)$ time.

2.2 *The C* mreps *package*

Our implementation of MKK algorithm takes about 6000 commented code lines. The basic package fits in 3500 lines, and can be used in other programs. Other files include interfaces, tests, and scripts. A substantial README is provided. At http://www.ens-lyon.fr/~magiraud/work, you can freely download all this package.

2.3 *Upgrading the implementation*

The main difficulty with the practical use of the MKK algorithm is the memory required for the DAWG construction. Its size is $O(n|\mathcal{A}|)$. However, since $\mathcal{A}$ is assumed to be finite and constant, its size is linear. With the 256-character alphabet, the program allocates 15 MB in order to compute the DAWG of a 20000-character word. For the 4-character alphabet {A, C, G, T}, the size decreases.

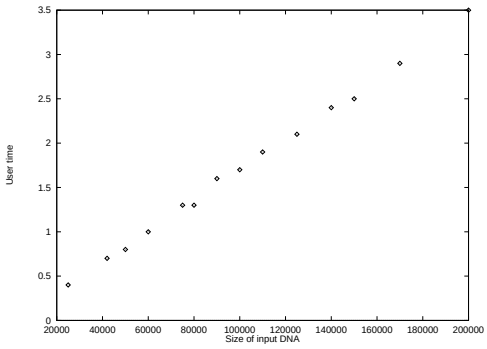
To reduce the memory, we use a sliding window of fixed size B, assuming that the repetitions have a bounded total length B. This assumption is reasonable in biological studies. Finding a more space-efficient structure, sufficient for our purposes, remains an interesting problem. Crochemore and V´erin's compacted DAWG [6] could be useful.

2.4 *Results of* mreps

```
pacman% reps atcacaaca
input : atcacaaca

s-fact : a t c a ca aca

from -(size)-> to : <per.> [exp.]
  5 -(2)-> 6 :      <1>    [2.00]  aa
  3 -(6)-> 8 :      <3>    [2.00]  acaaca
  2 -(4)-> 5 :      <2>    [2.00]  caca
-> 3 maximal repetitions.
```



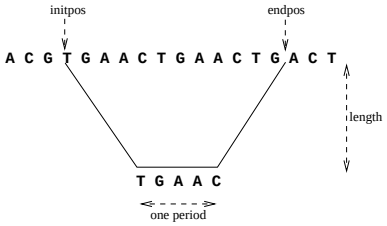
We tested mreps on very long *Fibonnacci words*, for which we know the exact number of repetitions (see [10]). We observed the linear-time behaviour of the program.

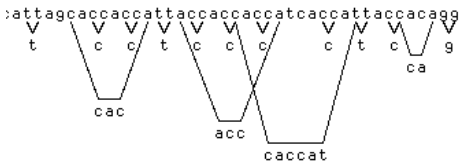
2.5 *Vizualizing repetitions*

Sequence landscapes have been proposed in [2] to visualize distant repetitions. Clift draws this landscape using one triangle by occurrence of a pattern. Inspired by this idea, we designed and implemented a method for visualizing successive repetitions.

We have designed a way to represent repetitions. With the graf program, one can display graphically maximal repetitions. Each repetition is shown by a trapezoid, whose height h is related

to the total length of the repetition whereas the base b shows one copy of the repeating pattern.





3. Application to genomic sequences

3.1 Repetitions in biology

Eukaryotic DNA sequences contain many repetitions. They can be successive, or not, and represent **duplications** of entire fragments, especially in telomer zones, to smaller **satellites**, which are used for marking the sequence (as $(CA)_n$ for Genethon maps).

3.2 Results of mreps

We ran mreps on several complete genomic sequences, in particular :

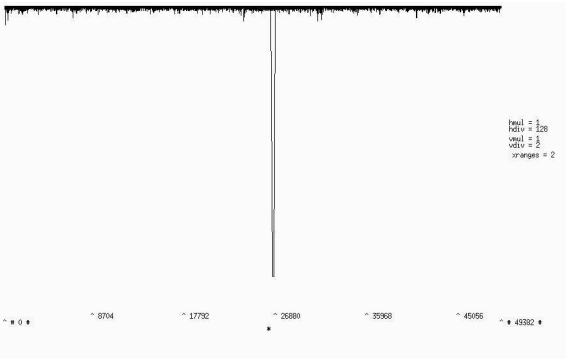
- Two prokaryotes : bacterians **E. coli** [1] ($4,7 \cdot 10^6$ bases) and **B. subtilis** [2] ($4,2 \cdot 10^6$ bases)
- One eukaryote : **yeast** [3] *(levure)* (16 chromosoms, $12 \cdot 10^6$ bases).

Corresponding results can be found of the web at http://www.ens-lyon.fr/~magiraud/work. Computation times were approximatively *15 minutes* for 4 millions bases with a sliding window of length 20000 on an old SPARC at 110MHz.

Example : On the first chromosom of yeast, you can find these two big repetitions, which are conjugated ! The figure shows the graf output visualizing one of those repetitions.

```
   26425 -(420)-> 26844 :      <135>   [3.11]
GTAGAGGTAAAAGTGTCGTTCCATGGCTGAGTTGTAGTCATGGCAGTAGT
GGCTGTTGTTGGTGTTCTGATGACAATGATGGTCTCATCAGTTGGCAAAC
CATTGGTACCGGTGACTGTGGTCAATTCGGTAGAAGTAGAGGTAAAAGTG
TCGTTCCATGGCTGAGTTGTAGTCATGGCAGTAGTGGCTGTTGTTGGTGT
TCTGATGACAATGATGGTCTCATCAAGTTGGCAAACCATTGGTACCGGTG
ACTGTGGTCAATTCGGTAGAAGTAGAGGTAAAAGTGTCGTTCCATGGCTG
AGTTGTAGTCATGGCAGTAGTGGCTGTTGTTGGTGTTCTGATGACAATGA
TGGTCTCATCAGTTGGCAAACCATTGGTACCGGTGACTGTGGTCAATTCG
GTAGAAGTAGAGGTAAAAGTG

   204512 -(420)-> 204931 :      <135>   [3.11]
CACTTTTACCTCTACTTCTACCGAATTGACCACAGTCACCGGTACCAATG
GTTTGCCAACTGATGAGACCATCATTGTCATCAGAACACCAACAACAGCC
ACTACTGCCATGACTACAACTCAGCCATGGAACGACACTTTTACCTCTAC
TTCTACCGAATTGACCACAGTCACCGGTACCAATGGTTTGCCAACTGATG
AGACCATCATTGTCATCAGAACACCAACAACAGCCACTACTGCCATGACT
ACAACTCAGCCATGGAACGACACTTTTACCTCTACTTCTACCGAATTGAC
CACAGTCACCGGTACCAATGGTTTGCCAACTGATGAGACCATCATTGTCA
TCAGAACACCAACAACAGCCACTACTGCCATGACTACAACTCAGCCATGG
AACGACACTTTTACCTCTAC
```



1. obtained on http://www.genetics.wisc.edu
2. from ftp://ftp.pasteur.fr/pub/GenomeDB/SubtiList
3. from ftp://ftp.mips.embnet.org/pub/yeast/

Acknowledgements

We would like to thank all members of POLKA project at LORIA.

References

[1] BLUMER, A., BLUMER, J., HAUSSLER, D., EHRENFEUCHT, A., CHEN, M. T., AND SEIFERAS, J. The smallest automaton recognizing the subwords of a text. *Theoret. Comput. Sci. 40* (1985), 31–55.

[2] CLIFT, B., ET AL. Sequence landscapes. *Nucleic Acid Research 14*, 1 (1986), 141–158.

[3] CROCHEMORE, M. Recherche linéaire d'un carré dans un mot. *C. R. Acad. Sc. Paris 296* (1983), 781–784.

[4] CROCHEMORE, M. Transducers and repetitions. *Theoret. Comput. Sci. 45* (1986), 63–86.

[5] CROCHEMORE, M., AND RYTTER, W. *Text Algorithms*. Oxford University Press, 1994.

[6] CROCHEMORE, M., AND VÉRIN, R. On compact directed acyclic word graphs. *LNCS 1261* (1997), 192–211.

[7] GUSFIELD, D. *Algorithms on strings, trees, and sequences*. Cambridge University Press, 1997.

[8] KNUTH, D. E., MORRIS JR., J. H., AND PRATT, V. R. Fast pattern matching in strings. *SIAM J. Comput. 6* (1977), 323–350.

[9] KOLPAKOV, R., AND KUCHEROV, G. Finding maximal repetitions in a word in linear time. In *Proceedings of the 1999 Symposium on Foundations of Computer Science, New York (USA)* (1999), pp. 596–604.

[10] KOLPAKOV, R., AND KUCHEROV, G. On maximal repetitions in words. In *Proceedings of the 12-th International Symposium on Fundamentals of Computation Theory, 1999, Iasi (Romania)* (1999), vol. 1684 of *LNCS*, pp. 374–385.

[11] LANDAU, G. M., AND SCHMIDT, J. P. An algorithm for approximate tandem repeats. *LNCS 684* (1993), 120–132.

[12] MAIN, M. G. Detecting leftmost maximal periodicities. *Discrete Applied Mathematics 25* (1989), 145–153.

[13] MAIN, M. G., AND LORENTZ, R. J. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms 5* (1984), 422–432.

[14] RODEH, M., PRATT, V. R., AND EVEN, S. Linear algorithm for data compression via string matching. *Journal of the ACM 28*, 1 (1981), 16–24.

[15] SAGOT, M.-F., AND MYERS, E. W. Identifying satellites in nucleic acid sequences. In *RECOMB99* (1999), pp. 234–242.