Lossless filter for multiple repetitions with Hamming distance

Pierre Peterlongo [a,*,1], Nadia Pisanti [b,1], Frédéric Boyer [c], Alair Pereira do Lago [d], Marie-France Sagot [c,e,2]

[a] *IRISA / INRIA, CNRS, Campus de Beaulieu, 35042 Rennes Cedex, France*

[b] *Dipartimento di Informatica, Università di Pisa, 56127 Pisa, Italy*

[c] *INRIA Rhône-Alpes and Laboratoire de Biométrie et Biologie Évolutive, UMR 5558, Université Claude Bernard, Lyon, 69622 Villeurbanne, France*

[d] *Instituto de Matemática e Estatística Universidade de São Paulo, 05508-090 São Paulo, Brasil*

[e] *King's College, London, London WC2R 2LS, UK*

Abstract

Similarity search in texts, notably in biological sequences, has received substantial attention in the last few years. Numerous filtration and indexing techniques have been created in order to speed up the solution of the problem. However, previous filters were made for speeding up pattern matching, or for finding repetitions between two strings or occurring twice in the same string. In this paper, we present an algorithm called Nimbus for filtering strings prior to finding repetitions occurring twice or more in a string, or in two or more strings. Nimbus uses gapped seeds that are indexed with a new data structure, called a bi-factor array, that is also presented in this paper. Experimental results show that the filter can be very efficient: preprocessing with Nimbus a data set where one wants to find functional elements using a multiple local alignment tool such as Glam, the overall execution time can be reduced from 7.5 hours to 2 minutes.

Key words: approximate repetitions, k-factors, multiple local alignment, bi-factors, bi-factor array

1 Introduction

Finding approximate repetitions (motifs) in strings is one of the most challenging tasks in text mining. Its relevance grew recently because of its application to biological sequences. Although several algorithms have been designed to address this task, and have been extensively used, the problem still deserves investigation for certain types of repetitions. Indeed, when the latter are long and the number of allowed differences among them grows proportionally to their length, there is no exact tool that can efficiently manage to detect such repetitions. Widely used efficient algorithms for multiple alignment are heuristic, and offer no guarantee that false negatives are avoided. On the other hand, exhaustive inference methods cannot handle queries where the differences allowed among the occurrences of a motif are as many as $5-10\%$ of the length of the motif, and the latter is as "long" as, say, 100 DNA bases. Indeed, exhaustive inference is done by extending or assembling in all possible ways shorter motifs that satisfy certain conditions. When the number of differences allowed is relatively high, this can therefore result in too many false positives (of intermediate length) that saturate the memory. In this paper, we introduce a preprocessing filter, called NIMBUS, where most of the data containing such false positives are discarded in order to perform a more efficient exhaustive inference. Our filter is designed for finding repetitions distant pairwise by a maximal Hamming distance, occurring in $r \geq 2$ input strings, or occurring possibly more than twice in one string. To our knowledge, one finds in the literature filters for local alignment between two strings [23,17,15], or for approximate pattern matching [20,3] only (that is a different task). Heuristic methods such as BLAST [1,2] and FASTA [16] filter the input data and extend only *seeds* that are repeated short fragments satisfying some constraints. NIMBUS is based on similar ideas but uses different requirements concerning the seeds; among the requirements are frequency of occurrence of the seeds, concentration and relative position. Similarly to [17,15], we use also a concept related to gapped seeds that has been shown in [4] to be particularly efficient for pattern matching. The filter we designed is lossless: unlike BLAST or FASTA, NIMBUS guarantees not to discard any repetitions meeting the input parameters. It uses necessary conditions based on combinatorial properties

* Corresponding author. Tel +332.99.84.74.59 Fax: +332.99.84.71.71
Email addresses: pierre.peterlongo@irisa.fr (Pierre Peterlongo), pisanti@di.unipi.it (Nadia Pisanti), frederic.boyer@inrialpes.fr (Frédéric Boyer), alair@ime.usp.br (Alair Pereira do Lago), Marie-France.Sagot@inria.fr (Marie-France Sagot).

[1] Supported by the ACI Nouvelles Interfaces des Mathématiques *π-vert* project of the French Ministry of Research.
[2] Supported by the ACI Nouvelles Interfaces des Mathématiques *π-vert* project of the French Ministry of Research, the ARC *BIN* project from the INRIA, and the ANR project *REGLIS*.

of multiple and approximate repetitions, and an algorithm that checks such properties in an efficient way. The efficiency of the filter relies on an original data structure, the *bi-factor array*, that is also introduced in this paper, and on a labelling of the seeds similar to the one employed in [8]. This new data structure can be used to speedup other tasks such as the inference of structured motifs [19] or for improving other filters [14].

It is worth mentioning that NIMBUS is meant as a preprocessing to *any* tool that searches for long motifs with mismatches shared by several strings, or that performs multiple alignments based on the detection of such repetitions. The goal of NIMBUS is to reduce the input size for these tools in order to obtain an overall significant speed up. For example, we shall see in Section 6 that preprocessing a data set where one wants to find a multiple local alignment with a tool such as GLAM ([7]), the overall execution time can be reduced from 7.5 hours to 2 minutes. The latter shortened time includes both the running time of the filtering of NIMBUS, and that of the tool applied to the filtered strings. Notice that we chose GLAM as a representatively good software for multiple alignment, but the filter could be used as preprocessing step for any other multiple alignment algorithm.

A preliminary version of this paper appeared in [22].

2 Necessary Conditions for Long Repetitions

A *string* is a sequence of zero or more symbols from an alphabet Σ. A string s of length n on Σ is represented also by $s[0]s[1]\ldots s[n-1]$, where $s[i] \in \Sigma$ for $0 \leq i < n$. The length of s is denoted by $|s|$. We denote by $s[i,j]$ the *substring*, or *factor*, $s[i]s[i+1]\ldots s[j]$ of s. In this case, we say that the string $s[i,j]$ occurs at position i in s. We call k-factor a factor of length k. If $s = uv$ for $u,\ v\ \in \Sigma^*$, we say that v is a *suffix* of s.

Definition 1 *Given r input strings $s_1, \ldots, s_r$, a length L, and a distance d, we call a (L,r,d)-**repetition** a set $\{\delta_1, \ldots, \delta_r\}$ such that $0 \leq \delta_i \leq |s_i| - L$, and for all $i,j \in [1,r]$ we have that*

$$d_H(s_i[\delta_i, \delta_i + L - 1], s_j[\delta_j, \delta_j + L - 1]) \leq d.$$

where by d_H we mean the Hamming distance between two strings, that is, the minimum number of letter substitutions that transform one into the other.

Given m input strings, the goal is to find the substrings of length L that are repeated in at least $r \leq m$ strings with at most d substitutions between each pair of the r repetitions, with L, r and d given. In other words, we want to extract all the (L,r,d)-repetitions from a set of r strings among $m \geq r$ input strings. The goal of the filter is therefore to eliminate from the input strings as many positions as possible that cannot contain (L, r, d)-repetitions. Since in most interesting applications, the value of the parameter d can be as big as 10% of L, a direct inference of repetitions of length L with $L/10$ substitutions based on a traditional algorithm would be unfeasible due to the big number of approximate short patterns that could contribute to a longer repetition. The main idea of our filter is based on checking necessary conditions on the number of *exact* k-factors that a (L, r, d)-repetition must share. Since there are only a linear number of exact factors and they are easy to find, this results in an efficient filtering. A string w of length k is called a *shared* k-factor for $s_1, \ldots, s_r$ if for all $i \in [1, r]$ we have that w occurs in s_i. Obviously, we are interested in shared k-factors that occur within substrings of length L of the input strings. Let p_r be the minimum number of non-overlapping shared k-factors that a (L, r, d)-repetition must have. It is intuitive to see that a $(L, 2, d)$-repetition contains at least $\left\lfloor \frac{L}{k} \right\rfloor - d$ shared k-factors, that is, $p_2 = \left\lfloor \frac{L}{k} \right\rfloor - d$. We now compute the value of p_r for $r > 2$.

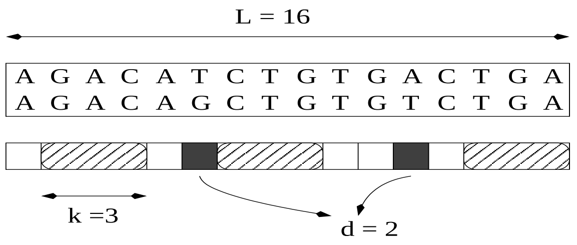
L = 16
A G A C A T C T G T G A C T G A
A G A C A G C T G T G T C T G A
k =3
d = 2

Fig. 1. Example of stained positions and shared k-factors for $r = 2$. Two strings of length $L = 16$ distant by $d = 2$ substitutions: the number of stained positions (represented by black squares) is 2. Let us consider, say, $k = 3$ for which we have that $\left\lfloor \frac{L}{k} \right\rfloor - d = \left\lfloor \frac{16}{3} \right\rfloor - 2 = 3$. The figure shows indeed one of the worst case scenario in which the placement of the stains limits the number of shared non-overlapping k-factors (represented by hashed rectangles) to 3.

Before stating the lemma, we give an intuition for the key observations that sustain the proof. Let us consider the (virtual) alignment of r strings of length L corresponding to an (L, r, d)-repetition. If they were identical, they would clearly share $\left\lfloor \frac{L}{k} \right\rfloor$ non-overlapping k-factors. Let us now assume that the Hamming distance between any two of the r strings is d. This is the worst case scenario for the minimum number of shared k-factors when all allowed substitutions are present, which is what we need to compute. If at position i (with $0 \leq i \leq L - 1$), there is a substitution between any pair of strings among the r, then we say that position i is *stained.* In this case, no k-factor involving this position can be shared by all the r strings. The worst case scenario applies when the stained positions (or *stains* for short) are spread in such a way that they eliminate distinct (i.e. non overlapping) k-factors, that is we loose as many potential $\left\lfloor \frac{L}{k} \right\rfloor$ non-overlapping k-factors as the number of stained positions in the alignment. We therefore need only to count the maximum number of stains. If $r = 2$, this number is simply d, and indeed we have that

$p_2 = \left\lfloor \frac{L}{k} \right\rfloor - d$. This case is shown in Figure 1. The key observation is that adding a third string (then a fourth, fifth, and so on up to the r^{th}) creates at most $\left\lfloor \frac{d}{2} \right\rfloor$ new stained positions. This is surprising because one could think that a single "new" string that presents d substitutions with each one of, say, $r-1$ previously considered strings, can stain up to $(r-1)d$ positions. Instead, it can stain only up to $\left\lfloor \frac{d}{2} \right\rfloor$ previously unstained positions independently from r (provided this is greater than 2). The reason is that, as we show below, many substitutions necessarily occur at positions that are already stained.

Let $\phi(r,d)$ denotes the maximal number of distinct positions that can be stained given that we have r strings such that any two of them have Hamming distance at most d. We have that $r \geq 2$ and clearly $\phi(2,d) = d$. In the following lemma we give tight numbers for $\phi(r,d)$.

Lemma 2 *Let $r \geq 2$ and $d \geq 0$ be integers. We have that $\phi(r,d) = \lceil d/2 \rceil + (r-1)\lfloor d/2 \rfloor$. In other words, $\phi(r,d) = \frac{rd}{2}$ if d is even and $\phi(r,d) = r\left\lfloor \frac{d}{2} \right\rfloor + 1$ if d is odd.*

PROOF. First, consider the following set of r sequences, all of them with length $\lceil d/2 \rceil + (r-1)\lfloor d/2 \rfloor$. Let i be any integer such that $1 \leq i \leq r$. All positions of a sequence s_i are `A`, except those in the following positions, which are `T`: the first $\lceil d/2 \rceil$ letters of s_1; the next $\lfloor d/2 \rfloor$ letters of s_2, instead of s_1; the next $\lfloor d/2 \rfloor$ letters of s_3, instead of s_2; and so on until the last $\lfloor d/2 \rfloor$ letters of s_r. More formally, $s_1[0] = \mathtt{T}$ and, for every $1 \leq i \leq r$ and $j > 0$, we set $s_i[j] = \mathtt{T}$ iff $\lceil d/2 \rceil + (i-2)\lfloor d/2 \rfloor \leq j < \lceil d/2 \rceil + (i-1)\lfloor d/2 \rfloor$; all other positions are set to `A`. One can verify that the Hamming distance from s_1 to any other sequence is d and the Hamming distance from any two sequences except s_1 is $2\lfloor d/2 \rfloor$. Since every position is a stain in these sequences, this proves that $\phi(r,d) \geq \lceil d/2 \rceil + (r-1)\lfloor d/2 \rfloor$ and that the provided bound is tight.

We now show that $\phi(r,d) \leq \lceil d/2 \rceil + (r-1)\lfloor d/2 \rfloor$. In order to prove it, we apply an induction on d. If, $d = 0$, $\phi(r,d)$ is clearly 0 and the proof is trivial. Since we are seeking an upper bound for $\phi(r,d)$, let us choose r sequences with the largest possible number of stains with the restriction that every pair of them has Hamming distance at most d. In order to evaluate the actual number $\phi(r,d)$ of stains, we have to consider two subcases. In the first case, no two of these sequences have Hamming distance exactly d, and hence we can state that $\phi(r,d) \leq \phi(r,d-1) \leq \lceil (d-1)/2 \rceil + (r-1)\lfloor (d-1)/2 \rfloor \leq \lceil d/2 \rceil + (r-1)\lfloor d/2 \rfloor$ due to the induction hypothesis. In the second case, at least one pair of sequences is at Hamming distance d. Without loss of generality, let us assume that - say - s_1 and s_2 are such that their Hamming distance is d. This leads to d stains at d such positions j where we have $s_1[j] \neq s_2[j]$. Let J be the set of such positions. If there is no other stain, the proof is finished since we have already proved that $\phi(r,d) = d = \lceil d/2 \rceil + \lfloor d/2 \rfloor \leq \lceil d/2 \rceil + (r-1)\lfloor d/2 \rfloor$. Suppose from now on that there is a stain $j' \notin J$. Hence, $s_1[j'] = s_2[j']$. By definition of a stain, there are two integers $i'' > i'$ such that $s_{i''}[j'] \neq s_{i'}[j']$. Hence, $i'' > 2$ and we can suppose that $3 \leq i'' \leq r$. (This in particular implies that $r \geq 3$.) If $i' \leq 2$, we take $i = i''$ and we have that $s_{i'}[j'] = s_1[j'] = s_2[j']$ differs from $s_i[j'] = s_{i''}[j']$. If $i' > 2$, we have that $s_1[j'] = s_2[j']$ differs either from $s_{i''}[j']$ or from $s_{i'}[j']$ and we can take $i \in \{i'', i'\}$ such that $s_1[j'] = s_2[j']$ differs from $s_i[j']$. Anyway, there is $i \in \{3, \ldots, r\}$ such that $s_1[j'] = s_2[j']$ differs from $s_i[j']$. For all $j \in J$, we have that $s_1[j] \neq s_2[j]$ and that $s_i[j] \neq s_1[j]$ or $s_i[j] \neq s_2[j]$. Hence, there is $i''' \in \{1, 2\}$ such that s_i differs from $s_{i'''}$ in at least $\lceil d/2 \rceil$ positions in J and also differs in j'. This implies that s_i cannot differ from $s_{i'''}$ in more than $d - \lceil d/2 \rceil = \lfloor d/2 \rfloor$ positions not in J. For $(r-2)$ possible values for i, we have $(r-2)\lfloor d/2 \rfloor$ possible values for j'. Hence, $\phi(r, d) \leq d + (r-2)\lfloor d/2 \rfloor = \lceil d/2 \rceil + (r-1)\lfloor d/2 \rfloor$.

The following theorem follows from Lemma 2.

Theorem 3 *Let $L > 0$, $r \geq 2$ and $d > 0$ be integers. The minimum number of non-overlapping shared k-factors that a (L, r, d)-repetition must have is*

$$p_r = \left\lfloor \frac{L}{k} \right\rfloor - d - (r-2) \times \left\lfloor \frac{d}{2} \right\rfloor .$$

PROOF. Since there are $\lfloor L/k \rfloor$ possible non-overlapping common factors in a (L, r, d)-repetition and that we have that any stain can hit at most one of them, then we have that $p_r = \left\lfloor \frac{L}{k} \right\rfloor - \phi(r, d)$, and hence the result follows.

Observe that the theorem applies also to the case where one is interested in finding (L, r, d)-repetitions occurring in a single string.

3 The Algorithm

We start by describing the algorithm for the case where we are looking for repetitions occurring in a set of strings. We call the corresponding algorithm Multi-Nimbus. Later, we describe how to modify the algorithm to find a (L, r, d)-repetition occurring in a single string thus obtaining an algorithm we call Mono-Nimbus.

Multi-Nimbus takes as input the parameters L, r and d, and m (with $m \geq r$) input strings. Given such parameters, it decides automatically the best value of k to apply Theorem 3 and compute the number of k-factors that are necessarily shared by a (L, r, d)-repetition.

The goal of Multi-Nimbus is to quickly and efficiently filter the strings in order to remove regions which cannot contain a (L, r, d)-repetition applying the necessary conditions described in Section 2 and keeping only the regions which satisfy these conditions. We compute the minimum number p_r of repeated k-factors each motif has to contain to possibly be part of a (L, r, d)-repetition. A set of p_r k-factors contained in a region of length L is called a p_r-set$_{\leq L}$. Multi-Nimbus searches for all the p_r-sets$_{\leq L}$ that are repeated in r of the m strings. All the positions where a substring of length L contains a p_r-set$_{\leq L}$ repeated at least once in r strings are kept by the filter, the others are rejected. The algorithm extracting the positions of all the p_r-sets$_{\leq L}$ is presented in Figure 2.

```
Multi-Nimbus_**Initialise**()
 1. **for** $g$ in $[(p_r - 2)k, L - 2k]$
 2.     **for** all $(k, g)$-bi-factors $bf$
 3.         Multi-Nimbus_Recursive($g - k$, positions($bf$), 2,
 4.                                  $firstKFactor(bf)$)
Multi-Nimbus_**Recursive** ($gmax$, $positions$, $nbKFactors$, $firstKFactor$)
 1. **if** $|Strings(positions)| < r$ **then** return // not in enough strings
 2. **if** $nbKFactors = p_r$ **then** save positions **and** return
 3. **for** $g$ in $[(p_r - (nbKFactors + 1)) \times k, gmax]$ // possible gaps length
 4.     **for** $(k, g)$-bi-factors $bf$ starting with firstKFactor
 5.         $positions$ = intersection($positions$, positions($bf$))
 6.         Multi-Nimbus_Recursive ($g - k$, $positions$, $nbKFactors + 1$,
 7.                                  $firstKFactor$)
```

Fig. 2. **Extract the positions of all the p_r-sets$_{\leq L}$**

To improve the search for the p_r-set$_{\leq L}$, we use what we call **bi-factors**, as defined below.

Definition 4 *A* (k, g)*-**bi-factor** is a concatenation of a factor of length k, a gap of length g and another factor of length k. The factor $s[i, i+k-1]s[i+k+g, i+2\times k+g-1]$ is a bi-factor occurring at position i in s. For the sake of simplicity, we also use the term* bi-factor *omitting k and g.*

For instance, the $(2, 1)$-bi-factor occurring at position 1 in $AGGAGAG$ is $GGGA$. The bi-factors occurring in at least r strings are indexed by means of a bi-factor array (presented in Section 4) that allows us to have access in constant time to the bi-factors starting with a specified k-factor. The main idea is to first find repeated bi-factors with a big gap g that may still contain $(p_r - 2)$ k-factors ($g \in [(p_r - 2)k, L - 2k]$). We call these *border bi-factors.* A border bi-factor is a 2-set$_{\leq L}$ that we then try to extend to a p_r-set$_{\leq L}$. To extend a i-set$_{\leq L}$ to a $(i+1)$-set$_{\leq L}$, we find a repeated bi-factor (called an *extending bi-factor*) starting with the same k-factor as the border bi-factor of the i-set$_{\leq L}$ and having a gap length shorter than all the other gaps of the bi-factors already composing the i-set$_{\leq L}$. The occurring positions of the $(i+1)$-set$_{\leq L}$ are the intersection of the extending bi-factor positions and of the positions of the i-set$_{\leq L}$. An example of this construction is presented in Figure 3.

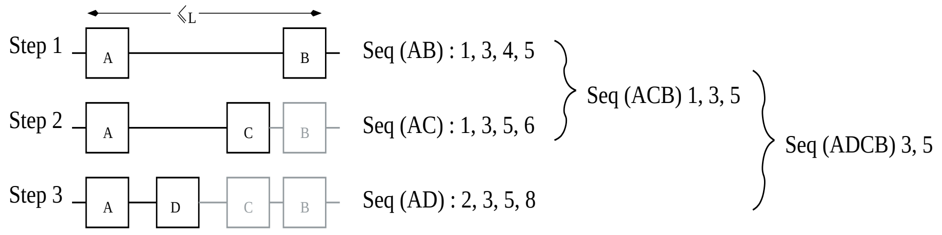

Fig. 3. **Example of the construction of a** 4**-set**$_{\leq L}$**.** In the first step, we find a bi-factor occurring at least once in at least $r = 2$ strings among $m = 8$ strings. During the second step, we add a bi-factor starting with the same k-factor (here called A), *included* inside the first one, and we intersect the positions. We repeat this once again (as $p_r = 4$) and obtain a 4-set$_{\leq L}$ occurring in strings 3 and 5. Actually, it is not only the string numbers that are checked during the intersection but also the positions in the strings, not represented in this figure for clarity.

In order to extract all the possible p_r-set$_{\leq L}$, we iterate the idea described above on all the possible border bi-factors: all bi-factors with gap length in $[(p_r - 2)k, L - 2k]$ are considered as a possible border of a p_r-set$_{\leq L}$. Furthermore, while extending a i-set$_{\leq L}$ to a $(i+1)$-set$_{\leq L}$, all the possible extending bi-factors have to be tested.

If the user runs Multi-Nimbus searching for very frequent repetitions (that is, r is large) with a high rate of substitutions (d is also large), the program has to choose a small value of k to avoid $p_r = 0$. However, a small value for k (*e.g.*, $k \leq 4$) leads to a slow and inefficient filter. In order to avoid this problem, we proceed in two steps as follows. In the first step, we start by running Multi-Nimbus assuming $r = 2$ (that is, with a weaker condition), which in general allows to increase the value of k while improving the sensitivity and the execution time. In the second step, the remaining strings are filtered using the initial parameters asked by the user. This results in an efficient strategy that we refer to as the **double pass** strategy. More in general, the best choice of the value of k and the consequent value of p_r changes from one application to another. Depending upon the parameters L, r and d as well as on the input size, there exist values of k that are not useful in practice, either because they are so small that they lead to a (too strong) condition which becomes hard to check, or conversely because they lead to a too weak condition (the reader can think of $p_r = 0$ as an extreme case) which is fast to check but useless. In between these cases, there is in general a range of “reasonable” values for k. This range in practice is not so big, as we show in Section 5.3 together with the performances of the filter the different values lead to. As we shall see, the best k to be used is hard to establish *a priori*, as one should find the best tradeoff between specificity and speed, which depends on the specific application. Thus, the choice for k may be set by the program or left to the user.

Finding (L, r, d)-repetitions inside a single string: MONO-NIMBUS MULTI-NIMBUS needs only a few modifications to obtain the MONO-NIMBUS algorithm finding (L, r, d)-repetitions occurring inside the same string instead of occurring distributed over $r \leq m$ strings. Theorem 3 remains true for a single string. The fact that a repetition is distributed over more than one string or not, does not change the value of p_r. The modifications to apply on the MULTI-NIMBUS algorithm in order to obtain MONO-NIMBUS are the following:

- index the bi-factors occurring at least r times (instead of occurring in at least r strings),
- the recursive part of the algorithm stops if the number of positions is lower than r (instead of the number of strings lower than r).

The **double pass** strategy is the same for MONO-NIMBUS as for MULTI-NIMBUS.

Complexity analysis

Since the complexity analysis applies either to MONO or to MULTI NIMBUS, we use the general NIMBUS name. Let us assume NIMBUS has to filter m strings[3], each of length ℓ. The total input size is then $n = \ell \times m$.

Memory analysis, average case. For each possible gap length of the bi-factors considered by the algorithm, a bi-factor array for one starting k-factor is stored in memory (taking in average $O(\frac{n}{|\Sigma|^k})$ space, as showed in Section (4)). The possible bi-factor gap lengths are in $[0,\ L - 2k]$. The total memory used by NIMBUS is therefore in $O(\frac{n}{|\Sigma|^k} \times L)$.

Time analysis, worst case. We here give an upper bound of the worst case time complexity. This bound is actually far from being tight in practice and is therefore a bit rough. Some future work should focus on improving this time complexity analysis, focusing on the average case and computing a more accurate complexity, possibly by means of an amortized analysis.

The overall worst case time complexity is in $O(L \times \ell \times n \times Z^{p_r - 1})$ with $Z = L \times \min(|\Sigma|^k, \ell)$. We shall see later (Fig. 6) that in practice computations

[3] In the case of MONO-NIMBUS, $m = 1$.

require much smaller time consumption.

4 The Bi-Factor Array

Since we make heavy use of the inference of repeated bi-factors, we have used a new data structure, called a **bi-factor array** (BFA), that directly indexes the bi-factors of a set of strings. This data structure is a modified suffix array. Indeed, in its original version [18], a suffix array does not permit to find objects such as bi-factors. We therefore adapted the algorithmic ideas developed for this data structure to create the bi-factor array. A bi-factor array is a suffix array adapted for (k, g)-bi-factors (with k and g fixed) that stores them in lexicographic order (without considering the characters composing the gap region). This data structure allows to access the bi-factors starting with a specified k-factor in constant time. NIMBUS never needs to consider at a same time bi-factors that start with different k-factors. Thus one BFA is constructed for each possible starting k-factor. Notice that the same data structure can be used to index bi-factors where the two factors have different sizes (say, (k_1, g, k_2)-factors with $k_1 \neq k_2$); we restrict ourselves here to the particular case of $k_1 = k_2$ because this is what we need for NIMBUS. For the sake of simplicity, we present the algorithm of construction of the bi-factor array for one string. The generalisation to multiple strings is straightforward.

We start by recalling the properties of a suffix array. Given a string s of length n, let $s[i \ldots]$ denote the suffix starting at position i. Thus $s[i \ldots] = s[i, n-1]$. The **suffix array** of s is the permutation π of $\{0,\ 1,\ \ldots,\ n-1\}$ corresponding to the lexicographic order of the suffixes $s[i \ldots]$. If $\underset{l}{\leq}$ denotes the lexicographic order between two strings, then $s[\pi(0) \ldots] \underset{l}{\leq} s[\pi(1) \ldots] \underset{l}{\leq} \ldots \underset{l}{\leq} s[\pi(n-1) \ldots]$. In general, another information is stored in the suffix array: the length of the **longest common prefix** (lcp) between two consecutive suffixes ($s[\pi(i) \ldots]$ and $s[\pi(i+1) \ldots]$) in the array. The construction of the permutation π of a text of length n is done in linear time and space [9][11][12]. A linear time and space lcp row construction is presented in [10].

In order to compute the BFA for bi-factors starting with a given k-factor using a suffix array and its lcp, we perform the following steps:

(1) Give every k-factor a **label**. For instance, in a DNA sequence with $k = 2$, AA has the label 0, AC has label 1 and so on. An array is created containing, for every suffix, the label of its starting k-factor. In the remaining of this paper, we call a ($label_1$, $label_2$)-bi-factor a bi-factor of which the two k-factors are denoted by $label_1$ and $label_2$.
(2) For each suffix, the label of the k-factor occurring $k + g$ positions before

the current position is computed and stored in a virtual[4] array we refer to as the predecessor label array.

(3) Construct the BFA for the bi-factor starting with the k-factor called $label_1$ as follows: the predecessor label array is traversed from top to bottom, each time the predecessor label value is equal to $label_1$, a new position is added to the part of the BFA where bi-factors start with $label_1$. Due to the suffix array properties, two consecutive bi-factors starting with $label_1$ are sorted with respect to the label of their second k-factor. The creation of the BFA is done such that for each ($label_1$, $label_2$)-bi-factor, a list of corresponding positions is stored.

We now explain in more detail how we perform the three steps above.

Labelling the k-factors. In order to give each distinct k-factor a different label, the *lcp* array is read from top to bottom. The label of the k-factor corresponding to the i^{th} suffix in the suffix array, called $label[i]$, is created as follows for $i \in [0, n-1]$:

$$label[i] = \begin{cases} 0 & \text{if } i = 0 \\ label[i-1] + 1 & \text{if } lcp[i] \leq k \\ label[i-1] & \text{ow} \end{cases}$$

Giving each suffix a predecessor label. For each suffix, the label of the k-factor occurring $k+g$ positions before has to be known. Let *pred* be the array containing the label of the predecessor for each position. It is filled as follows: $\forall\ i\ \in [0, |s|-1],\ pred[i] = label\Big[\pi^{-1}\Big[\pi[i] - k - g\Big]\Big]$ ($\pi^{-1}[p]$ is the index in the suffix array where the suffix $s[p\dots]$ occurs). Actually, the *pred* array is not stored in memory. Instead, each cell is computed on line in constant time. An example of the *label* and *pred* arrays is given in Figure 4.

Creating the BFA for a bi-factor starting with a k-factor called $label_1$. The BFA contains in each cell a ($label_1$, $label_2$)-bi-factor. We store the $label_1$ and $label_2$ values and a list of positions of the occurrences of the ($label_1$, $label_2$)-bi-factor. This array is constructed on the observation that for all i, the complete suffix array contains the information that a $(pred[i], label[i])$-bi-factor occurs at position $\pi[i]-k-g$. Traversing the predecessor array from top to bottom each time $pred[i] = label_1$, we either create a new $(label_1, label[i])$-bi-factor at position $\pi[i]-k-g$, or add $\pi[i]-k-g$ as a new position in the list

[4] In practice these values are not stored but computed on the fly.

i	lcp	π	associated suffix		*pred*	*label*
0	0	2	$AACCAC$		$\emptyset$	0
1	1	6	AC		1	1
2	2	0	$ACAACCAC$		$\emptyset$	1
3	2	3	$ACCAC$		1	1
4	0	7	C		4	2
5	1	1	$CAACCAC$		$\emptyset$	3
6	2	5	CAC		0	3
7	1	4	$CCAC$		3	4

Fig. 4. **Suffix array completed** with the *label* and the *pred* arrays for $k = 2$ and $g = 1$ for the text $ACAACCAC$.

of positions of the previous bi-factor if $label_2$ of the latter is equal to $label[i]$. An example of a BFA is given in Figure 5.

position(s)	($label_1$, $label_2$)	associated gapped-factor
0, 3	(1, 1)	$ACAC$
1	(3, 4)	$ACCC$

Fig. 5. **BFA of bi-factors starting with** AC. Here $k = 2$ and $g = 1$. The text is $ACAACCAC$. One can observe that the (2, 1)-bi-factor $ACAC$ occurs at two different positions.

The space complexity is in $O(n)$, as all the steps use linear arrays. Furthermore, no more than four arrays are simultaneously needed, thus the effective memory used is $16 \times n$ bytes. The first two steps are done in $O(n)$ time (simple traversals of the suffix array). The last step is an enumeration of the bi-factors found (no more than n). The last step is therefore in $O(n)$ as well. Hence the total time construction of the suffix array is in $O(n)$. With the following parameters: $L = 100$ and $k = 6$, Nimbus has to construct BFAs for around 90 different g values, which means 90 different BFAs. This operation takes, for strings of length 1Mb, around 1.5 minutes on a 1.2 GHz Pentium 3.

5 Testing the Filter

We tested a prototype of Nimbus on a 3 GHz Pentium 4 with 1Gb of memory. We performed several kinds of tests in order to evaluate how specificity and performances of the filter depend upon the input parameters. To this purpose, we used real biological sequences as well as randomly generated ones with *planted* repetitions. By "planted" repetitions, we mean that we manually inserted the repetitions in the strings so that we could know all the positions corresponding to a repetition, and hence evaluate how many positions, other than these, the filter (uselessly) keeps. The results of the tests made with randomly generated sequences are actually the average computed between several distinct runs (whose exact number changes from test to test since the variance was not always the same).

5.1 Speed and space with respect to input size

The first test is an evaluation of the actual time and space required by NIMBUS which, as we show here, is in practice better than the worst case analysis made in Section 3. Figure 6 shows the time and memory usage in function of the input data length and of the kind of data. One can observe that the space complexity is linear with the input length, although with a high multiplicative constant factor. The execution time depends on the type of sequences that are given as input to the algorithm. In Figure 6, we report the results obtained with the same parameters but with two distinct input files. On the left, the result refers to an experiment where the input data set consists of ten copies of the same string: in this way, all the positions must be kept by the filter, representing the worst case time complexity, which is a polynomial of degree p_r. On the right, the input strings are ten distinct strings, and the time complexity is clearly linear. We should bear in mind that these first results are from a prototype version of NIMBUS that could be improved to further reduce the time and space used.

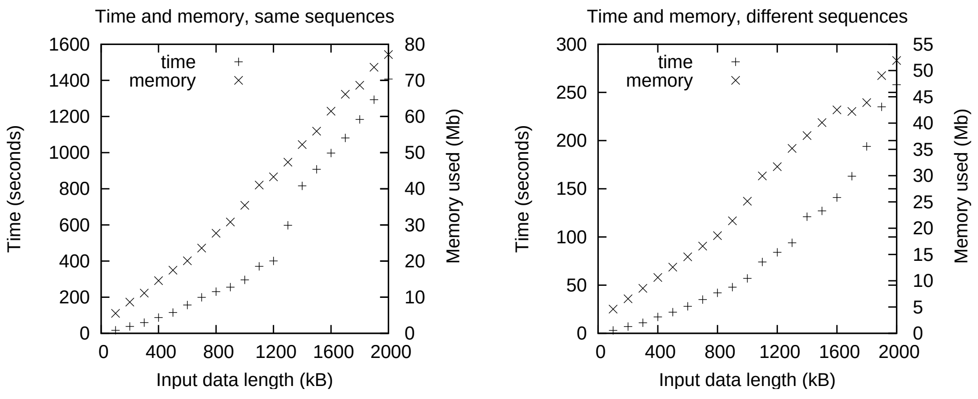

Fig. 6. Time and space spent by NIMBUS *w.r.t.* the input data length. The parameters are $L = 100$, $k = 6$, $d = 7$, $r = 3$ which implies $p_3 = 6$. The input file contains 10 strings of equal length. On the left, the same string is replicated 10 times and given as input, whereas on the right, we used 10 distinct random strings.

The second test aims at evaluating the performance of NIMBUS in terms of the percentage of input strings that are filtered, and how much remains thatdoes not corresponds to repetitions, while still keeping an eye on the time and space costs. These values are computed using $r = 2, 3, 4$ and four different types of input sequences: three randomly generated with different numbers of planted motifs (respectively $2, 5$, and 100), and one real biological sequence.

In general, both time and false positive ratios (that is, the ratio, computed on random strings with planted motifs, of non filtered data that are not part of a real motif) are low, showing a good performance of the filter.

In Figure 7, we present the behaviour of the filter for the four kinds of input strings of length 1 Mb. The first three strings are random and contain respectively 2, 5 and 100 motifs of length 100, pairwise distant by 10 substitutions. For each of these three strings, we ran NIMBUS in order to filter searching for motifs of length $L = 100$ occurring at least $r = 2$, 3 and 4 times with less than $d = 10$ substitutions. The last string is the genomic sequence of the *Neisseria meningitidis* strain MC58. *Neisseria* genomes are known for the abundance and diversity of their repetitive DNA in terms of size and structure [6]. The size of the repeated elements range from 10 bases to more than 2000 bases, and their number, depending on the type of the repeated element, may reach more than 200 copies. This fact explains why the *N. meningitidis MC58* genomic sequence has already been used [13] as a test case for programs identifying repetitive elements. We ran NIMBUS on this sequence in order to filter repetitions of length $L = 100$ occurring at least $r = 2$, 3 and 4 times with less than $d = 10$ substitutions.

For each input sequence, we compute the actual space costs and report (line "Memory Used") the constant multiplicative factor on the linear complexity.

For $r = 2$, we used $k = 6$ which gives a good result: around a few minutes execution time for all the random strings. One can observe that for the MC58 sequence, the execution time is longer (15 to 23 minutes) due to its high rate of repetitions.

For $r = 3$ and 4, to avoid a long and inefficient filter, we apply the double pass strategy described earlier, and start the filtration with $r = 2$ and $k = 6$. The time results are therefore divided into two parts: the time needed for the first pass and the one needed for the second pass. The time needed for the second pass is negligible with respect to the time used for the first one. This is due to the fact that the first pass filters from 89 % to 99 % of the string, thus the second pass works on a string at least 10 times shorter than the original one. This also explains why no extra memory space is needed for the second pass. For $r = 3$, the second pass uses $k = 5$ while for $r = 4$, the second pass uses $k = 4$. With $r = 4$ the necessary condition ($p_4 = 5$ shared 4-factors) is weaker than for $r = 3$ ($p_3 = 5$ shared 5-factors). That is why for MC58, more positions are kept while searching for motifs repeated 4 times, than for motifs repeated 3 times. Without using the double pass, for instance on MC58, with $r = 3$ the execution time is around 6 hours (instead of 15.9 minutes). The false positive ratio observed in practice is very low (less than 1.3 %). In general, many of the false positives occur around a (L, r, d)-repetition motif and not elsewhere in the strings.

Filtered Seq.		2 Motifs	5 Motifs	100 Motifs	MC58
Memory Used		29	30	30	30
$r = 2$	Time (Mn)	1.7	1.7	1.8	15
	Kept	436 (0.04 %)	1 090 (0.11 %)	22 474 (2.2 %)	129 085 (12.91 %)
	FP	0.02 %	0.06 %	1.25 %	unknown
$r = 3$	Time (Mn)	1.7 + 0	1.7 + 0.1	1.8 + 0.1	15 + 0.9
	Kept	0 (0 %)	1 085 (0.11 %)	22 445 (2.2 %)	93 794 (9.38 %)
	FP	0 %	0.06 %	1.24 %	unknown
$r = 4$	Time (Mn)	1.7 + 0	1.7 + 0.1	1.8 + 0.1	15 + 7.3
	Kept	0 (0 %)	1 086 (0.11 %)	22 456 (2.2 %)	113 369 (11.34 %)
	FP	0.0 %	0.06 %	1.25 %	unknown

Fig. 7. Nimbus **behaviour** on four types of strings while filtering in order to find $r = 2$, 3 and 4 repetitions. The first three strings are random and have length 1 Mb. They contain respectively 2, 5 and 100 motifs of length 100 distant pairwise by 10 substitutions. The last string is a segment of the genomic sequence of the strain MC58 of *Neisseria meningitidis* also of length 1 Mb. "FP" stands for False Positive ratio and the "Kept" lines give the number and percentage of positions kept.

5.3 Time and false positive rate with respect to k

We then evaluated the range of possible values to be chosen for k, and the relative performances these lead to. We generated three random sequences, each of length 50000, with planted $(1000, 3, 10)$-repetitions. We ran Nimbus with parameters $L = 100$, $r = 3$, $d = 10$, and $k = 3, 4, 5, 6, 7$. For $k = 7$, we have $p_3 = 0$ and thus the filter is useless. For $k = 3$ or less there are so many shared k-factors that the filter becomes too slow. Figure 8 shows the results for the other values.

As anticipated earlier in the paper (Section 3), the observed actual range of reasonable values of k is not big. In this test, we have three such values: from the smallest $k = 4$ which takes more than seven hours and has a very low false positive ratio, to the fastest (6 seconds) which leads also to a more sensitive filter. As a general remark, low values of k are advised when specificity is the issue, and the extra time it costs is anyway lower than that of the tool for

	k=4	k=5	k=6
Time (seconds)	25655	32	6
False positive	0.12	0.15	0.20
Memory	11,524	11,744	11,160

Fig. 8. Speed and false positive ratio for several values of k.

which the filter is preprocessing. On the other hand, higher values of k lead to a very fast filter which, in a negligible time, can already sensibly shrink the sequences. Finally, and not surprisingly, the memory taken by NIMBUS does not depend on k.

5.4 Time and false positive rate with respect to L

We now show how the performance of NIMBUS depends on the length L of the repetition. We generated three random sequences, each of size 50000, with planted $(L, 3, d = L/10)$-repetitions for several values of L, and we ran the filters with parameters L, $r = 3$, $d = L/10$ using $k = 5$. The results concerning time and specificity are shown in Figure 9.

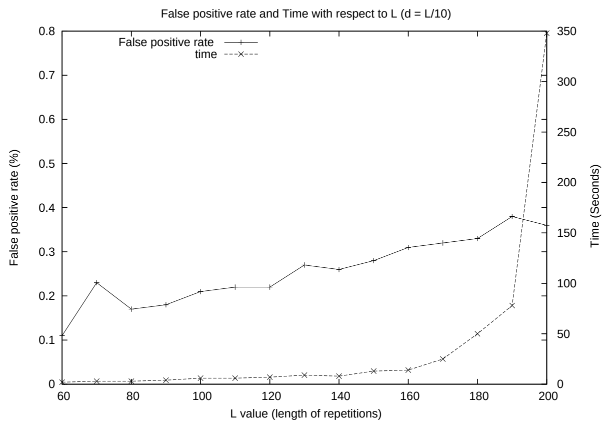

Fig. 9. Time and false positive ratio with respect to L.

We can observe that the false positive ratio is quite low (always below 0.4%), although it grows with L. The reason for the (slow) growth is the fact that when a portion of the input sequence satisfies the necessary condition, what is kept is indeed a fragment of length L. Time grows fast with respect to L. In this case, we may actually face an exponential time complexity with respect to p_r, which on its turn grows with L.

5.5 *Time and false positive rate with respect to d*

We now show the results of other runs where the tested parameter is d. We generated three random sequences, each of size 50000, and planted there ($L = 100, r = 3, d$)-repetitions with growing values of d. Hence, we ran NIMBUS with parameters $L = 100$, $r = 3$, using $k = 6$ with d from 0 to 13 (for higher values we have $p_3 = 0$). The results are shown in Figure 10.

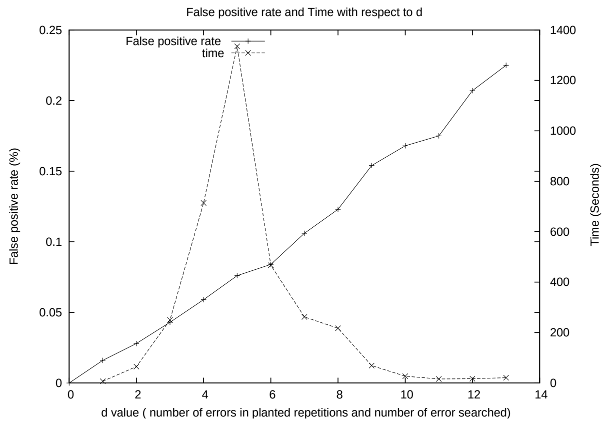

Fig. 10. Time and false positive ratio with respect to d.

We can observe that the false positive ratio, although remaining below 0.25, grows linearly with respect to d, due to the fact that the necessary condition becomes more and more weak. Concerning the time performance, the pick at $d = 5$ is quite interesting, especially because it was obtained in all runs for distinct randomly generated input strings. A possible explanation is that for this parameter set, $d = 5$ corresponds to an intermediate value between lower and higher ones. When the value of d is low, we have that p_r is so high (and hence the condition strong) that very little is kept and this requires a short time to be detected (very few shared k factors). When the value of d is high, p_r decreases and so does the time complexity (which asymptotically grows with p_r).

5.6 *Time and false positive rate with respect to r*

We now focus on the r parameter. We generated a set of R sequences. Each sequence was of length 10000 and contained an occurrence of a repetition of length 100 distant from each other by at most 3 substitutions. We let R vary from 2 to 10. For each value, we applied NIMBUS using the following parameters: $L = 100, d = 3, k = 5$ and $r = R$. We did not use the double pass strategy in order to focus only on the r parameter. The results are given in Figure 11. One can observe that the parameter r has not a big influence on time and specificity. Memory usage (data not shown) is also stable. One may however notice a light increase in the execution time for $r = 3$. This is due to the fact that, in this case, the necessary condition is high ($p_3 = 16$) and the number of recursive calls (see figure 2) is important.

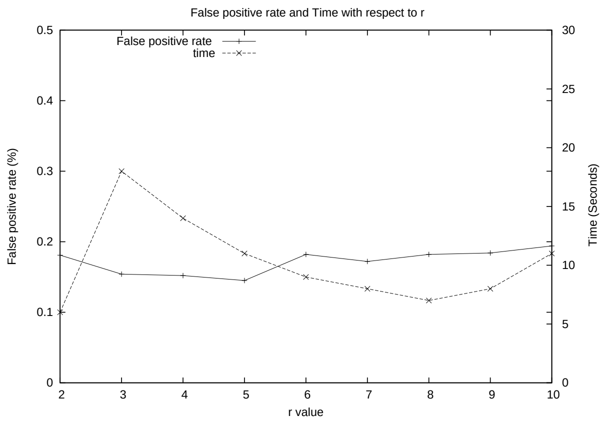

Fig. 11. Time and false positive ratio with respect to r.

However, the Figure may not accurately represent how the behaviour of NIMBUS varies with parameter r. Indeed, the previous test would not have been possible with, for instance, $d = 10$. Indeed in such a case, using $r = 4$ would have led to $p_4 = 0$.

Thus, the parameter r may have a strong influence, since increasing it can lead to a useless filter if the necessary condition p reaches zero. However, for *valid* ranges of r, its influence on the performance of NIMBUS is small.

6 Using the filter

In this section, we show two distinct possible applications of NIMBUS. The first concerns the inference of long biased repetitions, and the second multiple alignments.

6.1 Filtering for finding long repetitions

When inferring long approximate motifs, the number of differences allowed among the occurrences of a motif is usually proportional to the length of the motif. For instance, for $L = 100$ and allowing for as many as $L/10$ substitutions, one would have $d = 10$ which is high. This makes the task of identifying such motifs very hard and, to the best of our knowledge, no exact method for finding such motifs with $r > 2$ exists. Yet such high difference rates are common in molecular biology. The NIMBUS filter can efficiently be used in such cases as it heavily reduces the search space. We now show some tests that prove this claim. For testing the ability of Nimbus concerning the inference of long approximate repetitions, we ran an algorithm for extracting structured motifs called Riso [5] on a set of 6 sequences of total length 21 kB for finding motifs of length 40 occurring in every sequence with at most 3 substitutions pairwise. Using Riso, this test took 212 seconds. By previously filtering the data with Nimbus, the same test took 3.9 seconds. The filtering time was 0.1 seconds. The use of Nimbus thus enabled to reduce the overall time for extracting motifs from 212 seconds to 4 seconds.

6.2 Filtering for finding multiple local alignments

Multiple local alignment of r sequences of length n can be done with dynamic programming using $O(2^r n^r)$ time and $O(n^r)$ space. In practice, this complexity limits the application to a small number of short sequences. A few heuristics, such as MULAN [21], exist to solve this problem. One alternative exact solution could be to run Nimbus on the input data so as to exclude the non relevant information (i.e. parts that are too distant from one another) and then to run a multiple local alignment program. The execution time is hugely reduced. For instance, on a file containing 5 random strings of cumulated size 1 Mb each containing an approximate repetition [5], we ran Nimbus in approximately 1.5 minutes. On the remaining sequences, we ran a tool for finding functional elements using exact multiple local alignment called Glam [7]. This operation took about 25 seconds. Running Glam without the filtering, we obtained the same results [6] in more than 7.5 hours. Thus by using Nimbus, we reduced the execution time of Glam from 7.5 hours to less than 2 minutes.

Conclusions and future work

We presented a novel lossless filtration technique for finding long multiple approximate repetitions common to several strings or inside one single string. The filter localises the parts of the input data that may indeed present repetitions by applying a necessary condition based on the number of repeated k-factors the sought repetitions have to contain. This localisation is done us-

[5] Repetitions of length 100 containing 10 substitutions pairwise.

[6] Since Glam handles edit distance and Nimbus does not, in the tests we have used randomly generated data where we planted repetitions allowing for substitutions only, in order to ensure that the output would be the same and hence the time cost comparison meaningful.

ing a new type of seeds called bi-factors. The data structure that indexes them, called a bi-factor array, has also been presented in this paper. It is constructed in linear time. This data structure may be useful for various other text algorithms that search for approximate instead of exact matches. The practical results obtained show a huge improvement in the execution time of some existing multiple sequence local alignment and pattern inference tools, by a factor of up to 100. The theoretical complexity presented in the paper is a rough upper bound of the worst case. The results show that in practice the time consumption is much smaller than the theoretical complexity. Future work thus includes obtaining a better analysis of this complexity.

Other important tasks remain, such as filtering for repetitions that present an even higher rate of substitutions, or that present insertions and deletions besides substitutions. One idea for addressing the first problem would be to use bi-factors (and the corresponding index) containing one or two mismatches inside the k-factors. In the second case, working with edit instead of Hamming distance implies only a small modification on the necessary condition and on the algorithm but could sensibly increase the execution time observed in practice.

Acknowledgement

The authors would like to thank the anonymous referee for his careful reading and the very helpful comments that sensibly improved the paper.

References

[1] S.F. Altschul, W. Gish, W. Miller, E.M. Myers, and D.J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[2] S.F. Altschul, T.L. Madden, A.A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman. Gapped BLAST and PSI–BLAST: a New Generation of Protein Database Search Programs. *Nucleic Acids Research*, 25:3389–3402, 1997.

[3] S. Burkhardt, A. Crauser, P. Ferragina, H. P. Lenhof, and M. Vingron. q-Gram Based Database Searching Using a Suffix Array (QUASAR). In *RECOMB, Lyon*, 1999.

[4] S. Burkhardt and J. Karkkainen. Better Filtering with Gapped q-Grams. *12th Annual Symposium on Combinatorial Pattern Matching (CPM 2001)*, 56 of LNCS:51–70, 2001.

[5] A. M. Carvalho, A. T. Freitas, A. L. Oliveira, and M-F. Sagot. A Highly Scalable Algorithm for The Extraction of Cis-regulatory Regions. *Advances in Bioinformatics and Computational Biology*, 1:273–282, 2005.

[6] H. Tettelin *et al.* Complete Genome sequence of Neisseria Meningitidis Serogroup B Strain MC58. *Science*, 287(5459):1809–1815, 2000.

[7] M.C. Frith, U. Hansen, J. L. Spouge, and Z. Weng. Finding Functionnal Sequence Elements by Multiple Local Alignement. *Nucleic Acid Research*, 32(1):189–200, 2004.

[8] C. S. Iliopoulos, J. Mchugh, P. Peterlongo, N. Pisanti, W. Rytter, and M-F. Sagot. A First Approach to Finding Common Motifs With Gaps. *International Journal of Foundations of Computer Science*, 16(6):1145–1154, 2005.

[9] J. Kärkkäinen and P. Sanders. Simple Linear Work Suffix Array Construction. *International Colloquium on Automata, Languages and Programming (ICALP 2003)*, 2719 of LNCS:943–955, 2003.

[10] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time Longest-Common-Prefix Computation in Suffix Arrays and its Applications. *12th Annual Symposium on Combinatorial Pattern Matching (CPM 2001)*, 2089 of LNCS:181–192, 2001.

[11] D.K. Kim, J.S. Sim, H. Park, and K. Park. Linear-time Construction of Suffix Arrays. *14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, 2676 of LNCS:186–199, 2003.

[12] P. Ko and S. Aluru. Space Efficient Linear Time Construction of Suffix Arrays. *14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, 2676 of LNCS:203–210, 2003.

[13] R. Kolpakov, G. Bana, and G. Kucherov. MREPS: Efficient and Flexible Detection of Tandem Repeats in DNA. *Nucleic Acid Research*, 31(13):3672–3678, 2003.

[14] G. Kucherov, L. Noe, and M. Roytberg. Multiseed Lossless Filtration. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 02:51–61, 2005.

[15] M. Li and B. Ma. PatternHunter II: Highly Sensitive and Fast Homology Search. *Genome Informatics*, 14:164–175, 2003.

[16] D. J. Lipman and W. R. Pearson. Rapid and Sensitive Protein Similarity Searches. *Science*, 227:1435–1441, 1985.

[17] B. Ma, J. Tromp, and M. Li. Patternhunter: Faster and More Sensitive Homology Search. *Bioinformatics*, 18(3):440–445, 2002.

[18] U. Manber and G. Meyers. uffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948, 1993.

[19] L. Marsan and M.-F. Sagot. Algorithms for Extracting Structured Motifs Using a Suffix Tree With Application to Promoter and Regulatory Site Consensus Identification. *J. Comput. Bio.*, 7(3/4):345–360, 2000.

[20] G. Navarro, E. Sutinen, and Jorma Tarhio. Indexing Text with Approximate q-grams. *Journal of Discrete Algorithms*, 3(2-4):157–175, 2005.

[21] I. Ovcharenko, G.G. Loots, B.M. Giardine, M. Hou, J. Ma, R.C. Hardison, L. Stubbs, and W. Miller. Mulan: Multiple-Sequence Local Alignment and Visualization for Studying Function and Evolution. *Genome Research*, 15:184–194, 2005.

[22] P. Peterlongo, N. Pisanti, F. Boyer, and M-F. Sagot. Lossless Filter for Finding Long Multiple Approximate Repetitions Using a New Data Structure, the Bifactor Array. *String Processing and Information Retrieval (SPIRE 2005)*, 3772 of LNCS:179–190, 2005.

[23] K. R. Rasmussen, J. Stoye, and E. W. Myers. Efficient q-Gram Filters for Finding All ϵ-Matches Over a Given Length. In *RECOMB'05*, pages 189–203, 2005.