Lossless Filter for Finding Long Multiple Approximate Repetitions Using a New Data Structure, the Bi-Factor Array

Pierre Peterlongo[1], Nadia Pisanti[2] ⋆, Frederic Boyer[3], and Marie-France Sagot[3,4] ⋆⋆

[1] Institut Gaspard-Monge, Universite de Marne-la-Vallée, France
[2] Dipartimento di Informatica, Università di Pisa, Italy, and LIPN Université Paris-Nord, France
[3] INRIA Rhône-Alpes and LBBE, Univ. Claude Bernard, Lyon, France
[4] King's College, London, UK

Abstract. Similarity search in texts, notably 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 resolution of the problem. However, previous filters were made for speeding up pattern matching, or for finding repetitions between two sequences or occurring twice in the same sequence. In this paper, we present an algorithm called NIMBUS for filtering sequences prior to finding repetitions occurring more than twice in a sequence or in more than two sequences. 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* ([7]), the overall execution time can be reduced from 10 hours to 6 minutes while obtaining exactly the same results.

1 Introduction

Finding approximate repetitions (motifs) in sequences 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 quite long and the number of differences they contain among them grows proportionally to their length, there is no exact tool that can manage to detect such repetitions efficiently. 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

⋆ Supported by the ACI IMPBio *Evolrep* project of the French Ministry of Research.
⋆⋆ Supported by the ACI Nouvelles Interfaces des Mathématiques *π-vert* project of the French Ministry of Research, and by the ARC *BIN* project from the INRIA.

among the occurrences of a motif represent as many as $5-10\%$ of the length of the motif, and the latter is as small as, say, 100 DNA bases. Indeed, exhaustive inference is done by extending or assembling in all possible ways shorter motifs that satisfy certain sufficient conditions. When the number of differences allowed is relatively high, this can therefore result in too many false positives 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 in $r \geq 2$ input sequences, or repetitions occurring more than twice in one sequence. To our knowledge, one finds in the literature filters for local alignment between two sequences [21, 17, 15], or for approximate pattern matching [19, 3] only. Heuritic methods such as BLAST [1, 2] and FASTA [16] filter 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: contrary to the filter in BLAST or FASTA, ours does not discard any repetition meeting the input parameters. It uses necessary conditions based on combinatorial properties of 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 speed up other tasks such as the inference of structured motifs [18] or for improving other filters [14].

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 $s[0]s[1]\dots s[n-1]$, 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*, $s[i]s[i+1]\dots s[j]$ 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 $s_1, \dots, s_r$, a length L, and a distance d, we call a (L, r, d)-**repetition** a set $\{\delta_1, \dots, \delta_r\}$ such that $0 \leq \delta_i \leq |s_i| - L$. For all $i \in [1, r]$ 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 sequences, that is, the minimum number of letter substitutions that transforms 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 and d given. In other words, we want to extract all the (L, r, d)-repetitions from a set of r sequences among $m \geq r$ input sequences. 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. The value for parameter d can be as big as 10% of L. The main idea of our filter is based on checking necessary conditions concerning the amount of *exact* k-factors that a (L, r, d)-repetition must share. A string w of length k is called a *shared* k-factor for $s_1, \ldots, s_r$ if $\forall i \in [1, r]$ 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 $\frac{L}{k} - d$ shared k-factors, that is, $p_2 = \frac{L}{k} - d$. The value of p_r for $r > 2$ is given in the following result whose proof is omitted due to space limitations. However, the intuition is that the positions where there are substitutions between each pair of sequences must appear clustered because if two sequences differ in a position, then a third sequence will, at this position, differ at least from one of the other two.

Theorem 1. *A (L, r, d)-repetition contains at least $p_r = \frac{L}{k} - d - (r-2) \times \lfloor \frac{d}{2} \rfloor$ shared k-factors.*

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

3 The Algorithm

NIMBUS takes as input the parameters L, r and d, and m (with $m \geq r$) input sequences. Given such parameters, it decides automatically the best k to apply in order to filter for finding the (L, r, d)-repetitions either inside one sequence or inside a subset of r sequences. In the following, we present the algorithm for finding (L, r, d)-repetitions in r sequences. The algorithm can be adapted in a straightforward way to the case of finding (L, r, d)-repetitions occurring in a single sequence.

The goal of NIMBUS is to quickly and efficiency filter the sequences 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$_{\leq L}$. NIMBUS searches for the p_r-sets$_{\leq L}$ repeated in r of the m sequences. All the positions where a substring of length L contains a p_r-set$_{\leq L}$ repeated at least once in r sequences are kept by the filter, the others are rejected. To improve the search for the p_r-set$_{\leq L}$, we use what we call **bi-factors**, as defined below.

Definition 2. *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 simplicity's sake, we also use the term* bi-factor *omitting* k *and* g*.*

For example, the $(2, 1)$-bi-factor occurring at position 1 in $AGGAGAG$ is $GGGA$. The bi-factors occurring in at least r sequences are stored in 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 the biggest allowed 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 union of the extending bi-factor positions and of the positions of the i-set$_{\leq L}$. An example of this a construction is presented in Figure 1.

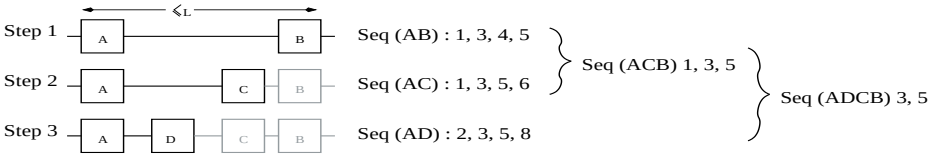

Fig. 1. Example of the construction of a 4-set$_{\leq L}$. At the first step, we find a bi-factor occurring at least once in at least $r = 2$ sequences among $m = 8$ sequences. During the second step, we add a bi-factor starting with the same k-factor (here called A), *included* inside the first one, and we merge the positions. We repeat this once and obtain a 4-set$_{\leq L}$ occurring in sequences 3 and 5. Actually not only the sequence numbers are checked during the merging but also the positions in the sequences, 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. The complete algorithm is presented in Figure 2.

Depending on the parameters, the k value may be too small (≤ 4) leading to a long and inefficient filter. In this case, we start by running NIMBUS with $r = 2$, often allowing to increase the k value, improving the sensitivity and the execution time. At the end, the remaining sequences are filtered using the initial parameters asked by the user. This actually results in an efficient strategy that we refer to later as the **double pass** strategy.

3.1 Complexity analysis

Let us assume NIMBUS has to filter m sequences each of length ℓ. The total input size is then $n = \ell \times m$.

```
NIMBUS_Initialise()
1.  for g in [(p_r − 2)k, L − 2k]
2.      for all (k, g)-bi-factors bf
3.          NIMBUS_Recursive(g − k, positions(bf), 2, firstKFactor(bf))
NIMBUS_Recursive (gmax, positions, nbKFactors, firstKFactor)
1.  if nbSequences(positions) < r then return //not in enough sequences
2.  if nbKFactors = p then save positions and return
3.  for g in [(p_r − (nbKFactors + 1)) × k, gmax] // possible gaps length
4.      for (k, g)-bi-factors bf starting with firstKFactor
5.          positions = merge(position, positions(bf))
6.          NIMBUS_Recursive (g − k, positions, nbKFactors+1, firstKFactor)
```

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

For each possible gap length of the bi-factors considered by the algorithm, a bi-factor array is stored in memory (taking $O(n)$ as showed in section (4)). The bi-factor gap lengths are in $[0,\ L-2k]$. The total memory used by NIMBUS is therefore in $O(n \times L)$. Let us assume that the time needed by the recursive extraction part of the NIMBUS algorithm depends only on a number of factors denoted by *nbKFactors*. We call this time $T(nbKFactors)$. With this notation NIMBUS takes $O\left(L \times \ell \times T(2)\right)$. Furthermore $\forall\ nbKFactors < p$:

$$T(nbKFactors) = \underbrace{\underbrace{L}_{\text{gap length}} \times \underbrace{\min(|\Sigma|^k, \ell)}_{\text{extending bi-factors}}}_{\text{replaced by } Z \text{ in the following}} \times (\underbrace{n}_{merge} + T(nbKFactors+1))$$

$$\begin{aligned} T(2) &= Z \times (n + T(3)) = Z \times n + Z \times T(3) \\ &= n \times Z + Z \times (Z \times n + Z \times T(4)) = n \times (Z + Z^2) + Z^2 \times T(4) \\ &\ \ \vdots \\ &= n \times \textstyle\sum_{i=1}^{p-2} Z^i + Z^{p-2} T(p) = n \times \frac{Z^{p-1}-Z}{Z-1} + Z^{p-2} \qquad (T(p) = O(1)) \\ T(2) &= O(n \times Z^{p-1}) \end{aligned}$$

The total time is therefore in $O(L \times \ell \times n \times Z^{p-1})$ with $Z = L \times \min(|\Sigma|^k, \ell)$. However, as we shall see later (Fig. 5), we have that this is just a rough upper bound of the worst-case. For instance, we do not take into consideration the fact that $T(i)$ decreases when i increases because of the possible decrease in the gap lengths. Furthermore, a *balance* exists between lines 4 and 5 of the recursion algorithm. For instance, if the sequences are composed only by the letter A, lines 4 and 5 will do only one merge but for n positions (in time $O(n)$). On the other hand, if the sequences are composed by n different letters, lines 4 and 5 will do n merges each in constant time, thus these two lines will be executed in time $O(n)$ as well. There can thus be a huge difference between the theoretical complexity and practical performance. The execution time strongly

depends on the sequences composition. For sequences with few repetitions, the filter algorithm is very efficient. See Section 5 for more details.

Finally, we have that creating the bi-factor arrays takes $O(L \times n)$ time which is negligible w.r.t. to the extraction time.

4 The Bi-Factor Array

Since we make heavy use of the inference of repeated bi-factors, we have designed a new data structure, called a **bi-factor array** (BFA), that directly indexes the bi-factors of a set of strings. The bi-factor array is a suffix array adapted for bi-factors (with k and g fixed) that stores them in lexicographic order (without considering the characters composing their gaps). This data structure allows to access the bi-factors starting with a specified k-factor in constant time. 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); 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 sequence. The generalisation to multiple sequences is straightforward. We start by recalling the properties of a suffix array. .

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

BFA construction. We start by listing the ideas for computing the BFA using a suffix array and its lcp:

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. A row 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 called $label_1$ and $label_2$.
2. For each suffix, the label of the k-factor occurring $k + g$ positions before the current position is known.
3. Construct the BFA as follows: let us focus for instance on the bi-factors starting with the k-factor called $label_1$. 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 for the part of the BFA where bi-factors start with the label $label_1$. Due to the suffix array properties, two consecutive bi-factors starting with the label $label_1$ are sorted w.r.t. 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:

$$label[0] = 0$$
$$\forall\ i \in [1, n-1]:\ label[i] = \begin{cases} label[i-1] + 1 & \text{if } lcp[i] \leq k \\ label[i-1] & \text{else} \end{cases}$$

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 $s[p\ldots]$ 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 3.

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

Creating the BFA. 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 $\pi[i]-k-g$. Let us focus on one first k-factor, say $label_1$. Traversing the predecessor array from top to bottom each time $pred[i] = label_1$, we either create a new $(label_1 = pred[i], label[i])$-bi-factor at position $\pi[i]-k-g$, or add $\pi[i]-k-g$ as a new position in the list of positions of the previous bi-factor if the $label_2$ of the latter is equal to $label[i]$. Of course, this is done simultaneously for all the possible $label_1$. An example of a BFA is given in Figure 4.

position(s)	($label_1$, $label_2$)	associated gapped-factor
2	(0,3)	$AACA$
0, 3	(1, 1)	$ACAC$
1	(3, 4)	$ACCC$

Fig. 4. BFA. Here $k = 2$ and $g = 1$. The text is $ACAACCAC$. One can notice that the (2, 1)-bi-factor $ACAC$ occurs at two different positions.

Complexity for creating a BFA The space complexity is in $O(n)$, as all the steps use linear arrays. Furthermore, one can notice that no more than four arrays are simultaneously needed, thus the effective memory used is $16 \times n$ bytes. The first two steps are in time $O(n)$ (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 sequences of length 1Mb around 1.5 minutes on a 1.2 GHz Pentium 3 laptop.

5 Testing the Filter

We tested NIMBUS on a 1.2 GHz Pentium 3 laptop with 384 Mb of memory.

Figure 5 shows the time and memory usage in function of the input data length. We can observe that the memory usage is worse in the case of identical sequences. This is due to the fact that all the positions contain L repeated bi-factors stored in memory. Furthermore, when the sequences are identical, all the positions are kept by the filter, representing the worst time complexity case. On the other hand, when all the sequences are distinct, the complexity is clearly linear.

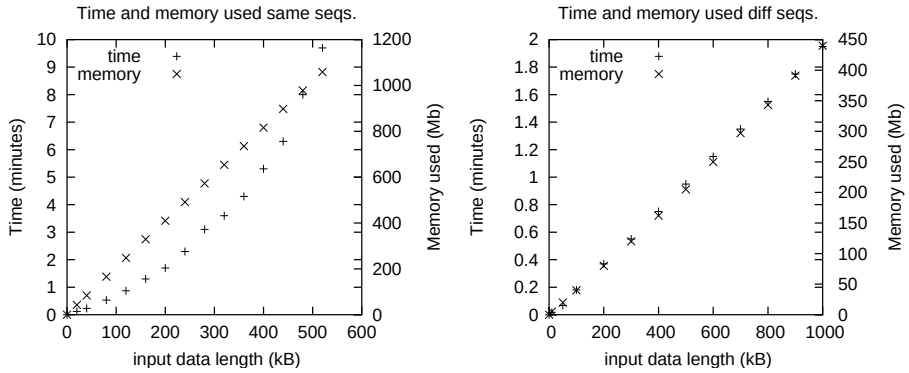

Fig. 5. Time and memory spent by NIMBUS *w.r.t.* the input data length. The parameters are $L = 100$, $k = 6$, $d = 7$, $r = 3$ which implies $p_3 = 6$. The input file contains 10 DNA sequences of equal length. On the left part of the figure the sequences are the same, whereas on the right part they are randomised.

In Figure 6, we present the behaviour of the filter for four kinds of input DNA sequences. The first three sequences are randomised and contain respectively 2, 5 and 100 motifs of length 100 distant pairwise by 10 substitutions. For each of these three sequences 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 DNA sequence 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 as a test case for programs identifying repetitive elements like in [13]. We ran NIMBUS on this sequence in order to filter the search for motifs of length $L = 100$ occurring at least $r = 2$, 3 and 4 times with less than $d = 10$ substitutions.

For $r = 2$, we used $k = 6$ which gives a good result: less than 5 minutes execution time for all the randomised sequences. On can notice that for the MC58 sequence, the execution time is longer (53 to 63 minutes) due to its high rate of repetitions.

For $r = 3$ and 4, we apply the double pass strategy described earlier, and start the filtration with $r = 2$ and $k = 6$. The time results are therefore subdivided 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 w.r.t. the time used for the first one. This is due to the fact that the first pass filters from 89 % to 99 % of the sequence, thus the second pass works on a sequence 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 $k = 4$, the efficiency of the filter is lower than for superior values of k. 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 memory used is 1435 Mb (instead of 943 Mb) and the execution time is around 12 hours (instead of 54 minutes). The false positive ratio observed in practice (that is, the ratio, computed on random sequences with planted motifs, of non filtered data that are not part of a real motif) is very low (less than 1.2 %). In general, many of the false positives occur around a (L, r, d)-repetition motif and not anywhere in the sequences.

Efficiency of the filter Although it depends on the parameters used and on the input sequences, the efficiency of the filter is globally stable. For instance, when asking for motifs of length $L \approx 100$ of which the occurrences are pairwise distant of $d \approx 10$, NIMBUS keeps also motifs of which the occurrences are pairwise distant up to $d + 7 \approx 17$. The smaller is d, the more efficient is NIMBUS. When $d = 1$ substitution, NIMBUS thus keeps motifs of which the occurrences are pairwise distant up to $d + 3 \approx 4$ only instead of $d + 7$ as for $d \approx 10$.

Sequence filtered		2 Motifs	5 Motifs	100 Motifs	MC58
Memory Used		675	675	681	943
$r = 2$	Time (Min.)	4.8	4.8	5	53
	Kept (Nb and Ratio)	406: 0.04 %	1078: 0.10 %	22293: 2.2 %	127782 : 12.7 %
	False Positive Ratio	0.02 %	0.08 %	2.0 %	unknown
$r = 3$	Time (Min.)	4.8 + 0	4.8 + 0.1	5 + 0.5	53 + 0.9
	Kept (Nb and Ratio)	0: 0 %	1078: 0.10 %	21751: 2.2 %	92069: 9.21 %
	False Positive Ratio	0 %	0.11 %	2.0 %	unknown
$r = 4$	Time (Min.)	4.8 + 0	4.8 + 0.1	5 + 0.5	53 + 10
	Kept (Nb and Ratio)	0: 0 %	1066: 0.11 %	21915: 2.2 %	106304: 10.63 %
	False Positive Ratio	0.0 %	0.09 %	1.8 %	unknown

Fig. 6. NIMBUS **behaviour** on four types of sequences while filtering in order to find $r = 2,\ 3$ and 4 repetitions

6 Using the filter

In this section we show two preliminary but interesting 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 230 seconds. By previously filtering the data with NIMBUS, the same test took 0.14 seconds. The filtering time was 1.1 seconds. The use of NIMBUS thus enabled to reduce the overall time for extracting motifs from 230 seconds to 1.24 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 $O(n^m)$ time and memory. In practice, this complexity limits the application to a small number of short sequences. A few heuristics, such as MULAN [20], 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 randomised sequences of cumulated size 1 Mb each containing an approximate repetition[5], we ran NIMBUS in approximatively 5 minutes. On the remaining sequences, we ran a tool for finding functional elements using multiple local alignment called GLAM [7]. This operation took about 15 seconds. Running GLAM without the filtering, we obtained the same results[6] in more than 10 hours. Thus by using NIMBUS, we reduced the execution time of GLAM from many hours to less than 6 minutes.

7 Conclusions and future work

We presented a novel lossless filtration technique for finding long multiple approximate repetitions common to several sequences or inside one single sequence. 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 using 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. Such results are in partial contradiction with the theoretical complexity presented in this paper. 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.

References

1. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. A basic local alignment search tool. *J. Mol. Biol.*, 215:403–410, 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 Res.*, 25:3389–3402, 1997.

[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.

3. S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. q-gram based database searching using a suffix array (quasar). In *proceedings of 3rd RECOMB*, pages 77–83, 1999.
4. S. Burkhardt and J. Karkkainen. Better filtering with gapped q-grams. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, number 2089, 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, Mar 2000.
7. M. C. Frith, U. Hansen, J. L. Spouge, and Z. Weng. Finding functional sequence elements by multiple local alignment. *Nucleic Acids Res.*, 32, 2004.
8. C. S. Iliopoulos, J. McHugh, P. Peterlongo, N. Pisanti, W. Rytter, and M. Sagot. A first approach to finding common motifs with gaps. *International Journal of Foundations of Computer Science*, 2004.
9. J. Karkkainen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. Assoc. Comput. Mach.*, to appear.
10. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, pages 181–192. Springer-Verlag, 2001.
11. D.K. Kim, J.S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, june 2003.
12. P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, to appear.
13. R. Kolpakov, G. Bana, and G. Kucherov. mreps: Efficient and flexible detection of tandem repeats in DNA. *Nucleic Acids Res*, 31(13):3672–3678, Jul 2003.
14. G. Krucherov, L.No, and M.Roytberg. Multi-seed lossless filtration. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching*.
15. M. Li, B. Ma, D. Kisman, and J. Tromp. Patternhunter ii: Highly sensitive and fast homology search. *J. of Comput. Biol.*, 2004.
16. D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity searches. *Sci.*, 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. 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. of Comput. Biol.*, (7):345–360, 2000.
19. G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate q-grams. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, number 1848 in Lecture Notes in Computer Science, pages 350–363, 2000.
20. 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.
21. K. R. Rasmussen, J. Stoye, and E. W. Myers. Efficient q-gram filters for finding all ϵ-matches over a given length. In *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching*, 2005.