Università di Pisa
Dipartimento di Informatica

Technical Report: TR-06-11

Lossless Filter for Long Multiple Repetitions with Edit Distance

Pierre Peterlongo[1] Nadia Pisanti[2] Alair Pereira do Lago[3]
Marie-France Sagot[4,5]

July 13, 2006

ADDRESS: via F. Buonarroti 2, 56127 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

Lossless Filter for Long Multiple Repetitions with Edit Distance

Pierre Peterlongo[1], Nadia Pisanti[2] ⋆, Alair Pereira do Lago[3], and Marie-France Sagot[4,5] ⋆⋆

[1] Institut Gaspard-Monge, Université de Marne-la-Vallée, France
[2] Dipartimento di Informatica, Università di Pisa, Italy
[3] Instituto de Matemática e Estatística Universidade de São Paulo
[4] INRIA Rhône-Alpes and Laboratoire de Biométrie et Biologie Évolutive, UMR 5558, Université Claude Bernard, Lyon, France
[5] King's College, London, UK

Abstract. Identifying local similarity between two or more sequences, or identifying repetitions occurring at least twice in a sequence, is an essential part in the analysis of biological sequences and of their phylogenetic relationship. Finding fragments that are conserved among several given sequences, or inside a unique sequence, while allowing for a certain number of insertions, deletions, and substitutions, is however known to be a computationally expensive task, and consequently exact methods can usually not be applied in practice. The filter we introduce in this paper, called Ed'Nimbus, provides a possible solution to this problem. It can be used as a preprocessing step to any multiple alignment method, eliminating an important fraction of the input that is guaranteed not to contain any approximate repetition. It consists in the verification of a strong necessary condition. This condition concerns the number and order of exactly repeated words shared by the approximate repetitions. The efficiency of the filter is due to this condition, that we show how to check in a fast way. The speed of the method is achieved thanks also to the use of a simple and efficient data structure, that we describe in this paper, as well as its linear time and space construction. Our results show that using Ed'Nimbus allows us to sensibly reduce the time spent in a local multiple alignment. The Ed'Nimbus software is freely available on the web: *http://igm.univ-mlv.fr/∼peterlon/officiel/ednimbus/*

1 Introduction

Multiple comparison is an essential part of the analysis of biological sequences and of their phylogenetic relationship. In particular, *local* multiple alignment is a powerful way to detect functional elements while identifying (multiple) repetitions inside a sequence is fundamental, for instance, in the investigation of transposition or duplication events. Finding fragments that are conserved among several given sequences or inside a unique sequence (*i.e.* finding *repetitions*) while allowing for a certain number of insertions, deletions, and substitutions, is however known to be a computationally expensive task, and consequently exact methods can usually not be applied in practice.

When comparing biological sequences in order to find features such as promoter or regulatory regions [2, 15], transposable elements or other types of long repetitions [17, 26], or to precisely identify the different rearrangements that may have happened [4, 7, 30], it is therefore important to be able to deal in a fast and accurate way with the huge amount of data represented by one or more whole genomes. Filtering, lossless or with possible loss, has appeared as one important way of dealing with part of the problem. The main idea behind it is to eliminate in a first, hopefully

⋆ Supported by the ACI Nouvelles Interfaces des Mathématiques *π-vert* project of the French Ministry of Research, the *ALGONEXT* project of the Italian Ministry of Research, and the *GALILEO* project of the Italian-French University.

⋆⋆ Supported by the ACI Nouvelles Interfaces des Mathématiques *π-vert* project of the French Ministry of Research, the ARC *BIN* project from the INRIA, the ANR project *REGLIS*, and the *GALILEO* project of the Italian-French University.

quick step, many portions of the input sequences before any further potentially costly investigation. Given a definition of what is the ultimate goal of the sequences comparison (for instance, it may be to identify all long repetitions of a certain type), lossless filters eliminate only those portions of the input data that are guaranteed not to contain the sought elements (in this example, the repetitions). Lossless filters produce therefore no false negatives. Filters that are not lossless cannot provide such guarantee and are thus heuristics.

In the last few years, there has been an increasing number of papers on the topic of filtering sequences prior to aligning them. This trend has been motivated by the fact that the problem of aligning biological sequences has scaled up considerably with the increasing number of genomes, notably of eukaryotes, that are being entirely sequenced and annotated. Filters, lossless or not, have thus been devised for comparing one string with itself [23], two strings pairwise [1, 6, 24, 25, 31, 32] or more than two strings among them at the same time [29]. Most filters rest on the idea that sequences that are reasonably similar have some parts that match exactly.

The fragments matching exactly that have been considered for building efficient filters have been, like in the Blast strategy [3], mostly single blocks (also called *seeds*) or a set of blocks separated by a fixed or flexible distance (also called *spaced-seeds*) as in [8–10, 24, 27, 29, 31, 32].

The authors of this paper have also contributed to the area in the past, notably in the context of identifying long repetitions allowing for substitutions only [29]. Results in terms of both specificity (that is, percentage of filtered data) and time were promising, but the tool Nimbus presented in [29] was meant as a lossless filter preceeding an alignment where no insertions or deletions were allowed, which is in general less relevant in biology.

In this paper, we eliminate this drawback by introducing a new filter, called Ed'Nimbus, that is lossless like Nimbus and has the same good performances while working with edit distance, that is, with insertions and deletions allowed besides substitutions. The other main characteristics of our filter, besides the fact that it is lossless and works within an edit distance framework, are that: 1. it is a filter that may be used, and indeed was conceived, for doing a multiple comparison, 2. it considers multiple seeds, and 3. it considers such multiple seeds taking into account both their number and the relative order among them.

Furthermore, Ed'Nimbus can take as input either m sequences or one unique sequence. In the first case, we want to find long approximate repetitions present in $r \leq m$ of the input sequences, and in the second case, we are looking for fragments occurring r times in the unique input sequence. The filter we present may however be used in more general contexts (for instance, as a first step in a multiple alignment algorithm) as we show with one application.

To the best of our knowledge, there is no lossless method that filters data for finding long multiple repetitions using edit distance. Among existing methods for finding repetitions, most use multiple seeds that have been shown in [9] to be particularly efficient for this task. Some deal with edit distance [1, 6, 8, 10, 27, 32] but for pairwise comparison only. Other methods [5, 6, 11, 20] quickly find multiple repetitions, but are heuristics and use a phylogenetic tree to drive the alignment. Hence, Ed'Nimbus is the first lossless filter for finding multiple repetitions within an edit distance framework.

It is worth stressing that Ed'Nimbus is meant as a preprocessing step to *any* method that searches for long approximate repetitions, or that performs a multiple alignment based on the detection of such repetitions. The goal of Ed'Nimbus is to reduce the input size for these methods in order to obtain a significant overall speed up.

The paper is organised as follows: we start by presenting the necessary condition we use for our lossless filter in Section 2. In Section 3, after having detailed the data structure used, we present the filtering algorithm and a complexity analysis. We then give in Section 4 the results of some tests: in Section 4.1 we show the specificity and time requirement of Ed'Nimbus, and in Section 4.2, we present some preliminary applications of Ed'Nimbus as a preprocessing of a multiple alignment.

Although Ed'Nimbus can identify repetitions either common to a set of sequences or that appear various times in a unique sequence, for clarity of exposition and because of space limitations, we address only, in Sections 2 and 3, the case of finding repetitions common to a set of sequences. The case of a unique sequence is treated in a very similar way and leads to the same performances.

2 Preliminary Definitions

A *sequence* is a set of zero or more symbols from an alphabet Σ. A sequence s of length n on Σ is represented also by $s[0]s[1]\dots s[n-1]$, where $s[i] \in \Sigma$ for $0 \le i < n$. The length of s is denoted by $|s|$. We denote by $s[i,j]$ the **word** $s[i]s[i+1]\dots s[j]$ of s. In this case, we say that the word $s[i,j]$ occurs at position i in s. We define a **k-factor** as a word of length k. A k-factor is said to be **shared** by two or more sequences if it occurs in all of them. We define a **k-hit** as a k-factor that is shared by at least two sequences. If $s = uv$ for $u,\ v\ \in \Sigma^*$, we say that u is a **prefix** of s and that v is a **suffix** of s.

Definition 1. *Given r sequences $s_1, \dots, s_r$, a length L, and a distance d, we call an (L,r,d)-**repetition** a set of integers $\{\delta_1, \dots, \delta_r\}$ such that*

$$\forall\ i \in [1,r] : 0 \le \delta_i \le |s_i| - L \text{ and}$$
$$\forall\ i,j \in [1,r] : d_E(s_i[\delta_i, \delta_i + L - 1], s_j[\delta_j, \delta_j + L - 1]) \le d.$$

The symbol d_E stands for the edit distance between two sequences, that is, the minimum number of letter insertions, deletions and substitutions that transforms one into the other.

Given m input sequences, let us suppose we want to find words of length L that occur in at least $r \le m$ sequences with at most d edit operations[1] between each pair of the r repetitions, where L, d and r are given by the user. In other terms, let us suppose we want to identify all the (L,r,d)-repetitions in a set of $r \le m$ input sequences. The goal of the Ed'Nimbus filter is to very quickly eliminate from the input sequences as many positions as possible that cannot belong to such repetitions. This operation can be seen as a preprocessing that drastically reduces the search space for any method that looks for approximate repetitions or performs local multiple alignments. Moreover, since (as we shall see in Section 4.1) Ed'Nimbus actually removes almost every sequence position that does not contain an (L,r,d)-repetition, it can also be used alone as a heuristic method with a very low false positive rate, and no false negatives (since it is lossless).

The main idea of our filter is based on checking a necessary condition concerning the amount of k-hits that an (L,r,d)-repetition must contain. Let p be the minimum number of possibly overlapping k-hits that an $(L,2,d)$-repetition must have. Pevzner and Waterman showed in [31] that:

$$p = L - (d+1) \times k + 1.$$

That is, if two words of length L have edit distance at most d, then they share at least $p = L - (d+1) \times k + 1$ k-factors.

Moreover, such k-hits obviously have also to occur in the same order in the two words of length L, even if not necessarily at the same distance from one another. In particular, a word of length L of the input can be part of an (L,r,d)-repetition only if it shares at least p k-hits in the same order, with words of length L that belong to at least $r-1$ other input sequences. Notice that this property is a weaker condition than having r or more words of length L that *pairwise* share at least p k-hits in the same order.

The idea of Ed'Nimbus is to check this (weaker) property for each possible starting position of a word of length L, and to discard those that do not satisfy this necessary condition.

3 Ed'Nimbus

The efficiency of Ed'Nimbus in checking the necessary condition we just indicated, is partly due to the use of a data structure, called the k-factor array. We start by presenting it, before giving the idea of the method (Section 3.2) and a detailed description of the algorithm (Section 3.3) with its complexity (Section 3.4).

[1] By *edit operations* we denote insertions, deletions and substitutions.

3.1 The k-factor array

The k-factor array is an array providing in constant time all positions in a text of a given k-factor. We show in this section how to construct this data structure.

Given a sequence s of length n, for $i \in [0, n-1]$, 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 words, 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 [18,21,22]. A linear time and space lcp row construction is presented in [19].

The k**-factor array** is a modification of the suffix array, to which we add two columns: the **label** and the **index**. A label is an integer $\in [0, |\Sigma|^k]$, and there is a label for each distinct k-factor occurring in the input sequences. All suffixes starting with the same k characters have the same label at the corresponding position in the array *label*, while all suffixes starting with different k characters have distinct labels. In order to give each distinct k-factor a different label, the *lcp* array is used. The label of the k-factor corresponding to the i^{th} suffix in the suffix array, called $label[i]$, is created as follows for each $i \in [1, n-1]$:

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

The second novelty of the k-factor is the **index** array, that allows to quickly retrieve the positions of all k-factors having a certain label. The *index* array is constructed such that $index[i]$ represents the index of the first occurrence of the suffix starting with a k-factor labelled i. The label and index arrays are created simultaneously in linear time and space. An example of a k-factor array is given in Figure 1.

i	lcp	π	associated suffix	*label*	*index*	
0	0	2	**AA**CCAC$	0	0	0
1	1	6	**AC**$	1	1	1
2	2	0	**AC**AACCAC$	1	2	4
3	2	3	**AC**CAC$	1	3	5
4	0	7	**C$**	2	4	7
5	1	1	**CA**ACCAC$	3		
6	2	5	**CA**C$	3		
7	1	4	**CC**AC$	4		

Fig. 1. The k**-factor array**, with the *label* and *index* array for the text ACAACCAC and $k = 2$.

In order to avoid problems due to k-factors occurring at one of the $k-1$ last positions of the text, one can consider that the text is followed by as many special symbols ($ for instance) alphabetically smaller than all others.

In the next section, we present an overview of the method of which we give a detailed description later in Section 3.3.

3.2 Overview of the method

ED'NIMBUS takes as input the parameters L, r and d, and m (with $m \geq r$) input sequences: $s_0, s_1, \ldots, s_{m-1}$. Given such parameters, first, if the user does not specify it, it fixes the value of

k to 6[2] to obtain the most suitable value of p. Secondly, this value is used to actually filter, that is, to detect portions of the input that satisfy the required condition, and hence that can contain an (L, r, d)-repetition. The goal of this section is to explain how the filtering works.

We remind that the condition that should be checked for each position of each sequence, is whether the word of length L starting at that position shares at least $p = L - (d+1) \times k + 1$ k-hits, that occur in the same order and within a window of size L, with at least $r - 1$ other input sequences. More formally, for each sequence s_j and for each one of its words $s_j[i, i+L-1]$ of length L (with $i \in [0, |s_j| - L]$), let $t(s_j, i)$ be the number of distinct sequences $s' \neq s_j$ where $\exists\, i' \in [0, |s'| - L]$ such that $s_j[i, i+L-1]$ and $s'[i', i'+L-1]$ have at least p k-hits occurring in the same order. We should compute $t(s_j, i)$ for each $j \in [1, m]$ and each $i \in [0, |s_j| - L]$, and keep only the positions that belong to the following set:

$$\{i \in [0, |s_j| - L] \;\mid\; j \in [1, m],\ t(s_j, i) \geq r - 1\}.$$

A first approach to do this would, for each sequence s_j in turn, use a single sliding window of size L that refers to a candidate repetition in s_j (we call this window the **reference window**) and, for this reference window, check whether its k-factors are k-hits shared, in the same order, with words of length L belonging to at least $r-1$ other input sequences. This first approach would therefore count, for each sequence s_j and each of its positions $i \in [0, |s_j| - L]$, and for each other sequence s' and each one of its positions $i' \in [0, |s'| - L]$, the number of k-hits of $s_j[i, i+L-1]$ and $s'[i', i'+L-1]$ that occur in the same order.

However, as is done in [8], in order to avoid considering all positions in all other sequences except s, we can divide those remaining sequences into blocks of length $2L$ (instead of L) that overlap in at least L positions[3], and then perform the following two steps:

1. Detect the blocks with which the reference window shares at least p k-factors (without looking at their relative order). We call such blocks the **good blocks**. This step is done using the k-factor array (as described in Section 3.1) that allows us to access in constant time the list of positions in the input sequences where a given k-factor occurs.
2. Keep, among the good blocks, only those that share at least p k-hits occurring in the same order. If these are in at least $r - 1$ distinct input sequences, then the position of the reference window is considered as a starting point for a possible repetition. Otherwise, it is rejected by the filter.

This second approach is the one adopted in ED'NIMBUS. We call the first approach NAIVE. Strictly speaking, this is a slight abuse of language since NAIVE is not a naive version of exactly the same algorithm as ED'NIMBUS because the two check different things.

Indeed, considering $O(N/L)$ blocks of size $2L$ rather that all $O(N)$ words of length L, leads to a factor L gain in the time complexity, but also in a loss of specificity. Since the resulting verified necessary condition is slightly weaker, we could keep parts of the sequences that would have been discarded by NAIVE. However, in Section 4.1, we compare the specificity and the time costs of ED'NIMBUS and of NAIVE in order to show that the small extra false positive ratio is by far compensated by the speed of ED'NIMBUS.

Before that, Section 3.3 describes the ED'NIMBUS algorithm in more detail.

3.3 The algorithm

In this section, we describe how exactly the filter works, showing which operations it performs in the initialisation step and each time the window is moved, and how the necessary condition is checked for each position of the window.

[2] The value of k is chosen according to an evaluation based on preliminary tests, that we do not describe in this paper. Notice, however, that the choice of k has possible effects on the efficiency (both in terms of time cost and filtration ratio) of the filter, but not on its correctness.

[3] That is, blocks of length $2L$ starting every L positions: each block overlaps both the previous and the next in L positions, hence ensuring that each word of length L is entirely included in at least one block.

The algorithm begins with the construction of a k-factor array as described in Section 3.1. A unique k-factor array is created for all the sequences as if they were concatenated, thus providing a unique and uniform labelling of the k-factors.

At each moment, the reference window refers to a word of length L in one of the input sequences, and all the other sequences are divided into overlapping blocks of length $2L$. We keep an array called *hits* that stores in $hits[b]$ the number of k-hits shared by the word of length L inside the window and the block b. Furthermore, we use an array called *nbGoodBlocks* having as length the number of input sequences, and containing in $nbGoodBlocks[i]$, for the reference windows, the number of good blocks in the sequence s_i. Both arrays are initially set to zero in all positions.

For each input sequence s_j, the window of length L is initially placed at position 0. In all the other sequences, we consider blocks of length $2 \times L$ that overlap by L positions, *i.e.* every L positions $(0, L, 2 \times L, \dots)$, a block of length $2 \times L$ starts. concerning the initial position of the window in the following way. For each k-factor (say, labelled l) found inside the reference window, for each block b_i where l occurs[4], $hits[b_i]$ is incremented by one. With this increment, if $hits[b_i]$ becomes equal to p, then b_i becomes a good block. In this case, if s_j is the sequence containing this good block, then $nbGoodBlocks[j]$ is incremented by one.

For each one of the other $O(N)$ positions of the reference window, we perform the following two steps:

1. Sliding the reference window:
When sliding the reference window, a k-factor is removed and another one is introduced. We pick the labels of both and read in the k-factor array the list of blocks they belong to; for the former, we decrease the corresponding entries in the array *hits*, while for the latter we increase them.

After decrementing its value, if $hits[b_i]$ becomes equal to $p-1$, then it means that b_i is no longer a good block. In this case, if s_j denotes the sequence containing b_i, then $nbGoodBlocks[j]$ is decremented by one. Symmetrically, we do as described above for all entries $hits[b_i]$ that are incremented.

2. Checking the relative order of the k-hits:
Using the *nbGoodBlocks* array, we check if at least $r-1$ sequences contain at least one good block. If this is the case, we check whether there are enough good blocks where the k-hits occur in the same order. Thus, for each good block b, we have to check whether at least p (out the $hits[b]$) k-hits are in the same order. This consists in computing the longest common subsequence (**LCS**) between the reference window and the considered block, after both have been rewritten using the alphabet of the labels associated with the k-factors[5]. If the LCS is greater or equal to p, then the condition is satisfied. More precisely, we proceed as follows. For each good block on a sequence where no good block has already been found containing at least p k-hits in the same order:

- Compute the LCS between this block and the reference window, both rewritten using the k-factor labels alphabet.
- If this is greater or equal to p, then mark the sequence where the block occurs as containing a possible repetition.

If at the end of the step, less than $r-1$ sequences contain a good block with at least p k-hits in the same order, then it means that the reference window cannot be an approximate repetition, and the corresponding position is discarded.

Notice that in this way, we check whether the word inside the current window has at least $r-1$ *possible matches* (that is blocks in which there are enough k-hits in the same order), but we do not check any similarity condition between pairs of such $r-1$ matches. This is a main reason for the speed of the filter.

We now list all possible cases that can lead to false positives, that is, all cases where a word of length L corresponding to a currently examined window is not discarded although it is not part of an (L, r, d)-repetition.

[4] This information is available in the k-factor array and accessible in constant time.

[5] In other words, at each possible position i of the sequence, we replace the letter at that position, say α, with the label representing the k-factor that begins at i.

First, it may happen that the reference window respects the necessary condition with $r-1$ blocs while the $r-1$ blocks are pairwise distant by more than d edit operations. Second, it can be that the block of size $2L$ does contain a real match, but that this has length greater than L. Third, a possible match may not be a real match: having p k-hits in the same order is only a necessary condition for two words of length L to be no more than d edit operations apart.

We can observe that the first case becomes more unlikely as the value of r is higher. The second case, apparently leading to many false positives, creates in fact only few. We show in Section 4 that in practical cases, the false positives due to this reason are rare. However, we show in Section 4 that the specificity of the filter is quite good in practice.

3.4 Complexity analysis

Let us assume ED'NIMBUS has to filter m sequences, each of length n, and let us recall that N denotes the total input size $n \times m$.

Space Complexity: The k-factor array construction can be done in linear time and space *w.r.t.* the data length (see Section 3.1). The *hits* array has size $\lfloor \frac{N}{L} \rfloor$ (that is the number of blocks). We use another array to store the results (the positions of the windows where a possible repetition occurs), which is simply an array of N booleans. Thus, the total space complexity is in $O(N)$, that is, linear in the input size.

Time complexity: We take into consideration two distinct cases. The first case, called *dense*, corresponds to sequences that contain a great number of highly similar repetitions. The second case, called *sparse*, corresponds to sequences containing only a few repetitions that are weakly similar. Furthermore, we assume that every possible k-factor exists in the sequences, considering that each k-factor occurs on average at $\frac{N}{|\Sigma|^k}$ positions instead of N positions. This assumption is realistic when filtering long DNA sequences (the goal of the filter) with small k-factors ($k \lessapprox 8$).

For each of the $O(N)$ positions of the sliding window, we do the following (we consider separately the two parts described in Section 3.3).

- *Sliding the window*: removing and adding one k-factor costs as much as the number of blocks of which the entry in the array k-hits has to be updated, that is, $O(\frac{N}{L \times |\Sigma|^k})$ both in the dense and in the sparse case. Notice that on sequences composed only of 'A' for instance, this complexity grows to $O(\frac{N}{L})$.
- *Checking the relative order of the k-hits*: this means computing the LCS of the windows against all the good blocks. Denoting by G the number of good blocks found for the current window, we thus have to compute G times the LCS between a word of length L against another of length $2L$. We compute the LCS using the method of [16] and described in [14] that takes time in $O(l \log l')$ to compute LCS(u,v), where l is the number of letters of u times the number of their occurrences in v, and l' is the length of LCS(u,v). In our case, l' is at most L, while l is at most $2 \times L^2$ (in the dense case), and can be assumed to be L in the sparse case.
  Summing up, in the dense case, this step takes $O(L^2 \times \log L \times G)$ time, whereas it takes $O(L \times \log L \times G)$ time in the sparse case.

Since we have to perform up to N times the two steps above, the overall time complexity is

$$O\Big(N \times (\underbrace{\frac{N}{L \times |\Sigma|^k}}_{\text{sliding}} + \underbrace{L^2 \times \log L \times G}_{\text{LCS}})\Big) = O\Big(\frac{N^2}{L \times |\Sigma|^k} + N \times L^2 \times \log L \times G\Big) \quad \text{(dense case)}$$

and

$$O\Big(N \times (\underbrace{\frac{N}{L \times |\Sigma|^k}}_{\text{sliding}} + \underbrace{L \times \log L \times G}_{\text{LCS}})\Big) = O\Big(\frac{N^2}{L \times |\Sigma|^k} + N \times L \times \log L \times G\Big) \quad \text{(sparse case)}$$

In the dense case, G can theoretically be as high as the total number of blocks, that is $\lfloor \frac{N}{L} \rfloor$. This leads to an overall complexity in $O(\frac{N^2}{L|\Sigma|^k} + N^2 \times L \times \log L)$. Notice, however, that it is enough to detect $r-1$ blocks that conserve the order of the k-factors and, in particular, that when one of them is found in a input sequence, no other good block of that sequence will have to be checked. Moreover, since we are in the dense case, we can expect that this happens soon enough to actually save many LCS computations with respect to the worst case in the complexity indicated above.

Comparatively with the time complexity in $O\Big(N^2 \times L + Y\Big)$ with $Y = N \times L^2 \times \log L \times G$ (dense case) or $Y = N \times L \times \log L \times G$ (sparse case) of the NAIVE algorithm mentioned in Section 3.2, the ED'NIMBUS algorithm represents a very good improvement, as confirmed by the tests shown in Section 4.1.

4 Experimental Results

In the next two sections, we show the time actually required by ED'NIMBUS on real biological sequences. As a first test (Section 4.1), we evaluate the growth of the execution time with respect to the input size, as well as the specificity with respect to time. Then, as a second test (Section 4.2), we show the use of the filter as a preprocessing step to a multiple local alignment algorithm. All tests were done on a 1.2 GHz Pentium 3.

4.1 Testing specificity and speed

We performed tests that show how much time ED'NIMBUS requires what is its specificity in practice.

For both evaluations (time and specificity), we compared the performance of ED'NIMBUS with respect to the NAIVE algorithm (Section 3.2). The specificity of NAIVE is higher than the one of ED'NIMBUS because the NAIVE algorithm checks the condition (Section 2) on words both of length L, while ED'NIMBUS does it on the current window (of length L) and blocks of length $2L$. ED'NIMBUS may thus detect false positives that NAIVE does not, leading to a weaker specificity. We show in this section that the difference in specificity is negligible, while the difference in speed is very high in favour of ED'NIMBUS.

We start with an experiment that aims to show how the execution time grows with respect to the input size. We generated for this 4 random DNA sequences of the same length, and we planted in each sequence a repetition of length $L = 100$ whose copies in the sequence have pairwise edit distance *exactly* equal to 10. We ran both ED'NIMBUS and NAIVE with parameters $L = 100$, $d = 10$, $r = 4$, $k = 6$. We did the same test for a length of the input sequences ranging from $n = 100$ to $n = 6000$, that is for a total input size, equal to $N = 4 \times n$, ranging from 400 to 24000. We could not go further because the NAIVE algorithm took too long, in particular since we repeated the experiment 50 times for each input size in order to report an average value. In Figure 2, we report the time taken by the two algorithms, expressed in seconds, with respect to the size of the input sequences. Observe that the difference is significant (the values are plotted in logarithmic scale).

The goal of this experiment was only to show how the time consumption grows with the input size for the two algorithms. We now show how the two methods behave in terms of specificity, still keeping an eye on the time costs. This is more relevantly done on real biological sequences. We thus considered the DNA 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 [12]. 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 [23]. On half of the sequence (about 1 Mb: more was not feasible for the NAIVE algorithm), we ran ED'NIMBUS, looking for repetitions of length $L = 100$, distant pairwise by at most $d = 10$ edit operation, occurring at least $r = 5$ times, using $k = 5$. It

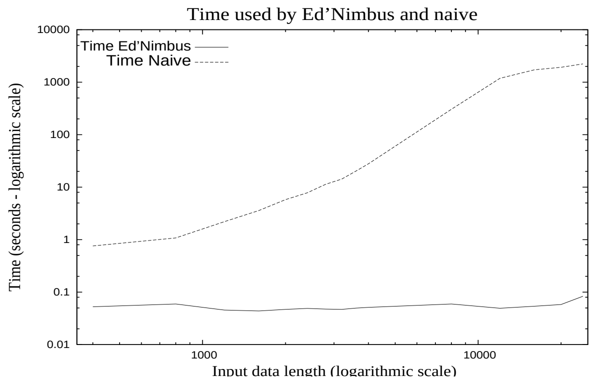

Fig. 2. Time spent by ED'NIMBUS and the naive algorithm *w.r.t.* the input size, in logarithmic scale. In all tests there are 4 input sequences of growing size (x axis), and we filter $(100, 4, 10)$-repetitions, with $k = 6$.

kept 7.3% of the input, after filtering, in about 8 minutes of computation. On the same sequence with the same parameters, the NAIVE algorithm kept 7.2% of the input, running for 7529 minutes (more than five days).

The results show that the ratio between time and specificity is very good for ED'NIMBUS on real DNA sequences.

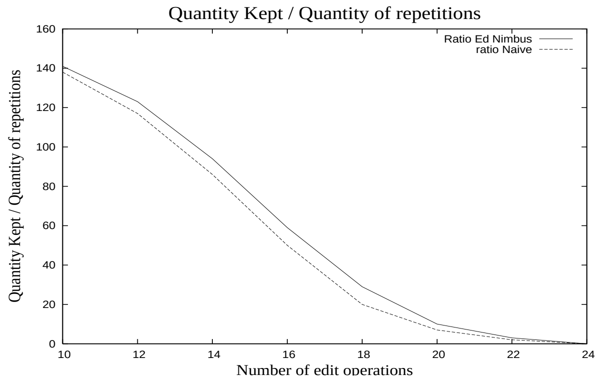

Fig. 3. Specificities of ED'NIMBUS and NAIVE. In all tests, there are 5 input random sequences (of length 10000), each containing a repetition of length 100 distant pairwise from 10 to 26 edit operations.

We then performed some tests on randomly generated sequences to check how both ED'NIMBUS and NAIVE behave when the input sequences contain repetitions with more edit operations than the limit specified by the user. We thus created a set of 5 sequences each containing a repetition of length 100, distant pairwise by 10, 12, 14, ..., 22, 24 edit operations. On those sequences, we ran ED'NIMBUS and NAIVE, filtering for repetitions of length $L = 100$, $r = 5$, $d = 10$ with $k = 5$. Each test was computed 50 times, the average results are given in Figure 3. The results show that, first, even if NAIVE has better specificity than ED'NIMBUS, the difference is low, and second, the filter is fully efficient even when the edit distance between the repetitions is 14 more than the limit specified by the user. Over this value, no repetition is detected anymore. However, one can notice that with these parameters, having 6 more edit operations leads to filtering ≈ 50 % of the repetitions.

We performed additional tests validating the good specificity of Ed'Nimbus coupled with a low time and memory usage. On one random sequence of length 2.2 Mb containing 3, 5, or 100 known planted repetitions with 10 edit operations pairwise, we ran Ed'Nimbus looking for (L, r, d) repetitions with r =3, 5, and 100, $L = 100$, $d = 10$ and $k = 6$. Furthermore, we performed the same tests on the MC58 sequence described above. Each test was computed 50 times, the average results are given in Figure 4. In this Figure, one can notice that on random DNA sequences of length 2.2 Mb, depending on the parameters, the time consumption is a few minutes and the memory usage is limited to less than 100 Mb.

Sequence filtered		3 Repetitions	5 Repetitions	100 Repetitions	MC58
Space Used (Mb)		71	70	72	82
$r = 3$	Time (Min.)	1.22	1.23	1.25	2.5
	Kept	0.01 %	0.02 %	0.48 %	15.5 %
	False Positive Ratio	0.006 %	0.01 %	0.25 %	unknown
$r = 5$	Time (Min.)	1.22	1.23	1.24	2.55
	Kept	0 %	0.01 %	0.24 %	9.11 %)
	False Positive Ratio	0 %	0.02 %	0.47 %	unknown
$r = 100$	Time (Min.)	1.22	1.01	1.4	4.3
	Kept	0 %	0 %	0.17 %	3.5 %
	False Positive Ratio	0 %	0 %	0.4 %	unknown

Fig. 4. Ed'Nimbus **behaviour** on four types of sequences of length 2.2 Mb, while filtering in order to find $r = 3,\ 5$ and 100 repetitions of length $L = 100$, distant pairwise by at most $d = 10$ edit operations.

4.2 Applications

We finally used Ed'Nimbus for the one of the main purposes we have designed it for: as a preprocessing step for performing a multiple local alignment. On m sequences of length n, this task can be done with dynamic programming using $O(2^m n^m)$ time and $O(n^m)$ space, where r is the number of sequences where a repetition must occur in order to contribute to the alignment. In practice, this is unfeasible for large and/or numerous sequences. As a consequence, the solutions proposed are mostly heuristics (as for example Mulan [28]) that do not guarantee to avoid false negatives. The alternative we suggest is to use any exact local alignment method after a preprocessing filtering of the input sequences done using Ed'Nimbus. The filtered data still contains the repetitions plus possibly a few other positions (as we have seen in Section 4.1), but it should now have become feasible, and much faster, to perform an exhaustive search for repetitions. Indeed, we expect that the preprocessing hugely reduces the overall execution time.

$GATA3$ is a gene that is involved in various development processes [28]. Since some such processes are identified, and differ among distinct species, it is interesting to perform a comparative analysis of this gene in different species by doing a multiple local alignment. We applied this idea to the $GATA3$ gene locus of 8 species (human, mouse, rat, chicken, frog, zebrafish, and tetraodon). The total size of the 8 input sequences is about 65000 bp.

We first ran Glam [13], a multiple local alignment method[6] to find repetitions of length 100, taking 10 minutes and 45 seconds. We then ran Ed'Nimbus on the same sequences with $L = 100$, $d = 8$, $r = 3$ and $k = 6$. It took only $7.e^{-10}$ minutes to output filtered data on which we ran Glam finding the same repetition as in the previous case in 24 seconds, that is, almost 27 times faster with a very fast preprocessing.

[6] Notice that we chose Glam as a representatively good software for multiple alignment, but the filter could be used as a preprocessing step to any other multiple alignment algorithm.

5 Conclusion

We presented in this paper a novel lossless filtration technique designed to precede any exact tool for finding long multiple approximate repetitions distant by a bounded number of substitutions, insertions and deletions. The repetitions may be common to several sequences or present inside one single sequence. As we have seen, the filter runs very fast and uses linear space to discard parts of the data that can not contain repetitions since they do not satisfy a necessary condition based on the number of exactly repeated words such repetitions must share and the way they are arranged. The speed of the method is achieved also thanks to the use of a simple and efficient data structure that is presented in the paper.

We showed in Section 4.1 that the specificity of the filter is very good. Moreover, we have described in Section 4.2 that using Ed'Nimbus allows us, on some preliminary applications, to apply exact multiple alignment methods in a short time.

Needless to say that Ed'Nimbus can also be used as a preprocessing step for a heuristic multiple alignment method, with the *only* purpose of speeding up the execution time.

As a next step of future work, we want focus on analisying how many among what we consider as false positives, are actually occurrences of what would be real repetitions is the constraint of all having the same length L would be relaxed. Indeed, flexible values for the length of the repetitions filtered, as done for instance in [32], would be interesting in practice, as well as natural for Ed'Nimbus. As a matter of fact, we believe that many of what we considered here false positives are actually occurrences of repetitions that have length, say, between L and $L+d$, actually realising the desiderable flexibility.

On this current version, Ed'Nimbus, available at *http://igm.univ-mlv.fr/~peterlon/officiel/ednimbus/* is useful either as a preprocessing step for any (multiple) local alignment, or, thanks to its high specificity, as a heuristic for finding long multiple repetitions.

References

1. S. A. Aghili, O. D. Sahin, D. Agrawal, and A. El Abbadi. Efficient Filtration of Sequence Similarity Search Through Singular Value Decomposition. In *Fourth IEEE Symposium on Bioinformatics and Bioengineering (BIBE'04)*, 2004.
2. N. Ahituv, S. Prabhakar, F. Poulin, E.M. Rubin, and O. Couronne. Mapping Cis-regulatory Domains in the Human Genome Using Multi-species conservation of synteny. *Human Molecular Genetics*, 14(20):3057–3063, 2005.
3. 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.
4. G. Bourque and P.A. Pevzner. Genome-Scale Evolution: Reconstructing Gene Orders in the Ancestral Species. *Genome Research*, 12(1):26–36, 2003.
5. N. Bray and L. Pachter. MAVID: Constrained Ancestral Alignment of Multiple Sequences. *Genome Research*, 14:693–699, 2004.
6. M. Brudno, C. B. Do, G. M. Cooper, M.F. Kim, E. Davydov, E. D. Green, A. Sidow, and S. Batzoglou. LAGAN and Multi-LAGAN: Efficient Tools for Large-Scale Multiple Alignment of Genomic DNA. *Genome Research*, 13:721–731, 2003.
7. M. Brudno, S. Malde, A. Poliakov, C. B. Do, O. Couronne, I. Dubchak, and S. Batzoglou. Glocal Alignment: Finding Rearrangements During Alignment. *Bioinformatics*, 19:54–62, 2003.
8. 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.
9. 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.
10. S. Burkhardt and J. Karkkainen. One-Gapped q-Gram Filters for Levenshtein Distance. *13th Annual Symposium on Combinatorial Pattern Matching (CPM 2002)*, 2373 of LNCS:225–234, 2002.
11. R. C. Edgar. MUSCLE: Multiple Sequence Alignment with High Accuracy and High Throughput. *Nucleic Acids Research*, Vol. 32, No. 5:1792–1797, 2004.
12. H. Tettelin *et al.* Complete Genome sequence of Neisseria Meningitidis Serogroup B Strain MC58. *Science*, 287(5459):1809–1815, 2000.

13. 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.
14. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
15. R.C. Hardison, F. Chiaromonte, D. Kolbe, Hao Wang, H. Petrykowska, L. Elnitski, S. Yang, B. Giardine, Y. Zhang, C. Riemer, S. Schwartz, D. Haussler, K.M. Roskin, R.J. Weber, M. Diekhans, W.J. Kent andM.J. Weiss, J. Welch, and W. Miller. Global Predictions and Tests of Erythroid Regulatory Regions. *Cold Spring Harbor Symp. Quant. Biol.*, 68:335–344, 2003.
16. J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *CACM*, 20(5):350–353, May 1977.
17. N. Juretic, T. E. Bureau, and R. M. Bruskiewich. Transposable Element Annotation of the Rice Genome. *Bioinformatics*, 20:155–160, 2004.
18. 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.
19. 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.
20. K. Katoh, K. Misawa, K. Kuma, and T. Miyata. Mafft: a novel method for rapid multiple sequence alignment based on fast fourier transform. *Nucleic Acid Research*, 30(14):3059–3066, 2002.
21. 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.
22. 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.
23. 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.
24. G. Kucherov, L. Noe, and M. Roytberg. Multiseed Lossless Filtration. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 02:51–61, 2005.
25. M. Li and B. Ma. PatternHunter II: Highly Sensitive and Fast Homology Search. *Genome Informatics*, 14:164–175, 2003.
26. M. Lipatov, K.D. Lenkov, D.A. Petrov, and C.M. Bergman. Paucity of Chimeric Gene-transposable Element Transcripts in the Drosophila Melanagaster Genome. *BMC Bioinformatics*, 24(3), 2005.
27. G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing Text with Approximate q-Grams. *11th Annual Symposium on Combinatorial Pattern Matching (CPM 2000)*, 1848 of LNCS:350–363, 2000.
28. I. Ovcharenko, G.G. Loots, B.M. Giardine, M. Hou, J. Ma, R.C. Hardison, L. Stubbs, and W. Millers. Mulan: Multiple-Sequence Local Alignment and Visualization for Studying Function and Evolution. *Genome Research*, 15:184–194, 2005.
29. P. Peterlongo, N. Pisanti, F. Boyer, and M-F. Sagot. Lossless Filter for Finding Long Multiple Approximate Repetitions Using a New Data Structure, the Bi-factor Array. *String Processing and Information Retrieval (SPIRE 2005)*, 3772 of LNCS:179–190, 2005.
30. P.A. Pevzner and G. Tesler. Genome Rearrangements in Mammalian Evolution: Lessons From Human and Mouse Genomes. *Genome Research*, 13:37–45, 2002.
31. P.A. Pevzner and M. Waterman. Multiple Filtration and Approximate Pattern Matching. *Algorithmica*, 13:135–154, 1995.
32. 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.