HAL open science

Rime: Repeat identification

Maria Federico, Pierre Peterlongo, Nadia Pisanti, Marie-France Sagot

To cite this version:

Maria Federico, Pierre Peterlongo, Nadia Pisanti, Marie-France Sagot. Rime: Repeat identification. Discrete Applied Mathematics, 2014, 163 (3), pp.275-286. 10.1016/j.dam.2013.02.016 . hal-00802023

HAL Id: hal-00802023

https://inria.hal.science/hal-00802023v1

Submitted on 29 Jun 2017

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

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

RIME: Repeat identification

Maria Federico [a], Pierre Peterlongo [b], Nadia Pisanti [c,d,*], Marie-France Sagot [e,f]

[a] *Dipartimento di Ingegneria dell'Informazione, University of Modena and Reggio Emilia, Italy*
[b] *INRIA Rennes - Bretagne Atlantique, EPI Symbiose, Rennes, France*
[c] *Dipartimento di Informatica, University of Pisa, Italy*
[d] *LIACS Leiden University, The Netherlands*
[e] *Université Lyon 1, CNRS, UMR5558, Laboratoire de Biométrie et Biologie Evolutive, Villeurbanne, France*
[f] *INRIA Grenoble Rhône-Alpes, France*

A B S T R A C T

We present an algorithm for detecting long similar fragments occurring at least twice in a set of biological sequences. The problem becomes computationally challenging when the frequency of a repeat is allowed to increase and when a non-negligible number of inser- tions, deletions and substitutions are allowed. We introduce in this paper an algorithm, RIME[1] (for Repeat Identification: long, Multiple, and with Edits) that performs this task,
and manages instances whose size and combination of parameters cannot be handled by other currently existing methods. This is achieved by using a filter as a preprocessing step, and by then exploiting the information gathered by the filter in the following actual repeat inference step. To the best of our knowledge, RIME is the first algorithm that can accurately deal with very long repeats (up to a few thousands), occurring possibly several times, and with a rate of differences (substitutions and indels) allowed among copies of a same repeat of 10–15% or even more.

1. Introduction

Many genomes, in particular of eukaryotes, contain a high proportion of repeats (for instance, nearly half of the human genome is covered by repeats [32]) whose copies (genomic *occurrences*) are approximate, and exhibit different lengths and characteristics depending on their origin and function. Two major classes of repeats are in general identified: tandem repeats of a length ranging from a few base pairs to a few hundred, and interspersed repeats that include transposons and retrotransposons both representing mobile genetic elements. Segmental duplications, which may involve thousands of base pairs, can be either tandem or interspersed. Until relatively recently, all such repeats, mainly occurring in the intergenic regions, were considered as *junk DNA*. However, our view on such elements is evolving fast. Transposons, for instance, are now believed to have a role in the immune system [13] and on gene regulation [22].

Depending on the species and the kind of repeats considered, the average number of occurrences of a repeat, its length and the rate of differences among the occurrences present a large variability. In this paper, we focus on the problem of finding long multiple repeats that may appear dispersed along a whole genome or chromosome, or that are common to different genomes/chromosomes. The proposed method is designed for identifying repeats that are multiple (at least two occurrences and in general more), long (typically of at least a hundred base pairs), and approximate (with up to 10–15% differences – substitutions, insertions or deletions – among the occurrences of a same repeated element).

* Corresponding author at: Dipartimento di Informatica, University of Pisa, Italy. Tel.: +39 0502213152; fax: +39 0502212726.
E-mail address: pisanti@di.unipi.it (N. Pisanti).

[1] RIME is also a reference to Coleridge's poem "The Rime of an Ancient Mariner" which contains many repetitions as a poetic device.

is high, represents a particularly difficult computational problem. Indeed, exact methods for finding multiple repeats use dynamic programming, leading to a time complexity exponential in the number of occurrences of the searched repeats. This is not practicable except on toy examples. For example, the three versions of a well-known exact algorithm (MSA [17]) compiled and hosted by the Pittsburgh Supercomputing Center can manage a limited number of sequences and sequence lengths (at most 50 sequences of up to 150 residues each, or 25 sequences of up to 500 residues, or 10 sequences of up to 1000 residues). There are many exact (and, in particular, non-lossy, meaning that they guarantee not to have false negatives) methods for finding approximate motifs, which are repeats of limited size and also in general highly conserved (e.g., [2,11,12,29,19]), but these do not scale to be applicable for finding long and biased repetitions and/or handling too frequent such repeats.

A broad range of algorithms for finding repeats is available. Some make use of *seeds* for anchoring the repeats before application of dynamic programming. They also often perform progressive alignments; that is, they combine pairwise alignments beginning with the most similar pair and then progressing to the most distantly related, following an order or a phylogenetic tree that must thus be given. Such approaches are heuristic and, when they are efficient, they do not guarantee completeness and sometimes not even accuracy of the results obtained. Some such methods are listed and commented in Section 4. However, despite the high relevance of this problem for its applications to the analysis of biological sequences, there is no satisfactory methodology available [1]. A recent promising approach [23–25] to combine efficiency and accuracy is to preprocess the data using a lossless filter. By lossless is meant a filter that enables one as quickly as possible to discard fragments of the input that cannot belong to any searched repeat. After such a filtering step, usually the remaining dataset will be much smaller than the original one, allowing the application of an accurate, although time-consuming, algorithm. The reader may refer to [27] for a state of the art survey on filters for sequence analysis.

In this paper we introduce Rime, a combinatorial approach that combines a filtering and an alignment step. The filter, called Tuiuiu, has been described previously [25,9], but it will be briefly recalled in Section 2. Tuiuiu is to date a state of the art filter for multiple repeats allowing also for indels. Rime uses (i) Tuiuiu as a preprocessing step and then (ii) the pieces of information collected by Tuiuiu for detecting only true repeats and finding their precise borders and locations, thereby completing the repeat inference task. The new algorithm is described in Section 3. Experiments and a discussion thereon are provided in Section 4.

A shorter and preliminary version of this work appeared in [10]. The software is available at the web page http://code.google.com/p/repeat-identification-rime/.

2. Preliminary definitions and recalling the Tuiuiu filter

A *string* is a concatenation 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 $|s|$, while, for a set S of strings, we denote with $|S|$ its length, being the sum of the lengths of all the strings it contains. We denote by $s[i, j]$ the *substring* $s[i]s[i+1]\ldots s[j]$ of s.

In this paper, we focus on the problem of finding (L, r, d)-*repeats*, which are repetitions of length L occurring at least r times and whose occurrences have pairwise edit distance bounded by d. Formally, (L, r, d)-*repeats* are defined as follows.

Definition 1 ((L, r, d)-*Repeat*)**.** Given a set $\mathcal{S}$ of one or more input strings, a length $L > 0$, an integer $r \geq 2$, and an edit distance $0 \leq d < L$, we call an (L, r, d)-*repeat* a set $\{\omega_1, \ldots, \omega_r\}$ of r words having length in the range $[L-d, L+d]$ occurring in the strings of $\mathcal{S}$ such that, for all $i, j \in [1, r]$, $d_E(\omega_i, \omega_j) \leq d$, where $d_E(\omega, \omega')$ is the edit distance between two strings ω and ω'.

The definition can be used to model repeats inside one string ($|\mathcal{S}| = 1$) or distributed among several strings ($|\mathcal{S}| > 1$). In the latter case, one can also enforce that the r words occur over r distinct strings (it must then be that $|\mathcal{S}| \geq r$). In both cases, when $r = 2$, the problem can be solved in quadratic time with dynamic programming. However, for multiple repeats, this solution is not feasible. The current exact exhaustive methods can manage input data of very limited size and/or are able to detect repeats for very small values of d only. On the other hand, heuristics do not guarantee to find all (L, r, d)-repeats. Filters, and in particular lossless ones, have been introduced with the goal of speeding up any method (exact or heuristic) by means of a drastic reduction in the size of the input which is obtained by eliminating most of the data not containing any repeat with the given parameters. This is useful for exact approaches to repeat detection, but also for heuristics as it enables one to gain in speed as well as possibly obtain results of better quality. In general, a filter is useful if it is much faster than the search step that it preprocesses (otherwise one would rather directly perform the search), and is at the same time as *selective* as possible, thus leaving the least number of false positives, corresponding to fragments of the input conserved by the filter and that turn out not to contain a repeat. The lossless filter Tuiuiu [25] was specifically designed to preprocess the inference of (L, r, d)-*repeats*, and indeed it takes as input the same parameters L, r, and d, as well as a set of one or more strings. The method consists in sliding a window w of length L along the input, checking whether there are at least $r-1$ other fragments with which w fulfills a strong and fast-to-check necessary condition for being at edit distance at most d from the window w. If this is the case, then w is kept; it is discarded otherwise. The windows taken into account are those starting at each possible position of the strings(s) ($O(n)$, where n is the length of the input string, or of the input set of strings),

while the fragments for which the condition is checked against w are overlapping blocks of size $L + b + d$ occurring every b positions, where b is the smallest power of 2 strictly greater than d. This choice, already adopted by previous filters [7,28], allows one to consider n/b blocks only, thus gaining in speed. Informally, the necessary condition employed by TUIUIU is to detect blocks that share enough small exact fragments of a given size q (called q-grams) with the current window, moreover occurring in the same order in the window and in the block. Our algorithm begins with using TUIUIU and then in several steps it shrinks the outcome of the filter in order to output all and only actual repeats. At the end, no (L, r, d)-repeats will be missed by RIME, while there can be (theoretically, as we actually had none in all our practical experiments) left some false positives with respect to the definition of (L, r, d)-repeats, which would be (as we show in Section 3) repetitions larger than L and less conserved. The efficiency and accuracy of RIME is due to the use of TUIUIU also as a guide for directly targeting sites of possible repetitions that are the blocks where the filter had detected windows that fulfilled the necessary condition. Formally, we have the following.

Definition 2 (*Friend Block of a Window*)**.** Given a window w of length L and a block B of length $L + b + d$, we say that B is a *friend* of w if B fulfills the condition of the filter TUIUIU.

It was proved in [25] that, if a window is part of an (L, r, d)-*repeat*, and a block B contains another occurrence of the same (L, r, d)-*repeat*, then B is a *friend* of w. The filter is thus lossless. Moreover, the size chosen for the blocks guarantees that any occurrence of an $(L, 2, d)$-repeat is always totally contained in at least one such block, and hence the filter remains lossless also after this choice.

On the other hand, given that what TUIUIU checks is just a necessary condition and not a sufficient one, then we may also have kept more than just (L, r, d)-*repeats*. This is the first possible source of false positives among the friends of a window, which we denote by FP_{cond}.

Furthermore, taking into account blocks rather than all possible fragments with a length in $[L-d, L+d]$, the selectivity of the filter becomes weaker as the necessary condition is checked against a block larger than the window, and strictly greater than $L + d$ which is the largest possible length for a fragment to be at edit distance at most d from a window of size L. In particular, nothing forbids that the q-grams that satisfy the condition and make a block B be a friend of a window w actually span over an interval of the input string that is too large to be part of an (L, r, d)-*repeat* although properly included in B. In other words, the fact that a window has a block as a friend does not necessarily mean that the block contains a fragment of size $L + d$ that fulfills the condition with w, and in this case the block is uselessly retained. This can thus be an additional source of false positives, which we denote by FP_{block}.

The necessary condition for two strings to be at edit distance at most d is then inserted in a suitable framework for detecting fragments of the input data that fulfill the requirement with respect to at least $r - 1$ others. If TUIUIU keeps a window w that has at least $r - 1$ blocks as friends, then there is reasonable hope that w and at least one fragment inside each one of the blocks are at an edit distance at most d from one another, but there is no guarantee that any two blocks contain at least one fragment each that are also at a pairwise distance at most d, or that they will all be kept by the filter. Indeed, any or both of the following cases can hold.

1. One or more pair(s) of the friends of w may not fulfill the necessary condition between them. In other words, w has enough blocks that are its friends but these do not contain windows that have friends in the other blocks. Should we represent friendship as an edge between nodes that correspond to genomic fragments, the condition checked can be seen as guaranteeing a star-shaped structure with w as center, while the requirement would actually be of a clique. This is another case of false positive, which we denote by FP^*.
2. It may turn out that a block that is friend of a window w is filtered out later during the filtering process because it does not contain any retained window. We call this an *empty block*. In this case, if too many blocks that are friends of w are empty, then w would be left with fewer than $r - 1$ non-empty blocks as friends and thus should have been discarded from further consideration.

Both cases can lead to w being a false positive should all these fragments be necessary for w to be part of an (L, r, d)-*repeat*. For this reason, TUIUIU performs an additional check for empty blocks, as well as multiple passes to remove all empty blocks, and thus (sensibly) reduce the number of such false positives with a very small extra time requirement. For more details about TUIUIU and its optimization, the reader can refer to [9,25]. In general, when using TUIUIU, we make a double pass as a default choice.

3. The algorithm RIME

RIME has two versions: one to detect repeats occurring in distinct input sequences (in which case the frequency requirement r must be at most as large as the number of sequences given as input, and its meaning is that the occurrence must occur in at least r distinct sequences), and the other to detect repeats occurring (without overlapping) at least r times in a single sequence. We now describe the algorithm RIME that is designed to exploit the information obtained by TUIUIU to find (L, r, d)-*repeats*. We start with some observations about windows contained in overlapping blocks, as the relation between windows and the blocks that contain them is critical in some steps of the algorithm, and also because we eventually merge overlapping blocks that turn out to contain a repeat in order to identify possibly longer repeats.

3.1. Overlapping blocks and blocks merging

We recall that the input string (strings) is (are) subdivided into blocks of size $L + b + d$ that start every b positions.

Definition 3 (*Consecutive Windows and Consecutive Blocks*)**.** Two windows are said to be *consecutive* is one starts at position i and the other at position $i + 1$ for some $1 \leq i \leq n - L$. Two blocks are said to be consecutive if their starting positions c and c' are such that $|c - c'| = b$.

Definition 4 (*Overlapping Blocks*)**.** Two blocks B and B', starting at positions c and c' respectively, are said to be *overlapping* (or to overlap) iff $|c - c'| < L - (b + d)$.

Two consecutive blocks, for example, are certainly overlapping and, in that specific case, they overlap by $L + d$ positions.

Observation 1. Given a string S, any word $w = S[p, p+L-1]$ of length L in S (with $p \leq n-L+1$) can be entirely contained in at most two consecutive blocks. In particular, if c is the starting position of a block, then we have that:

- if $p \in [c, c+d]$, then w belongs to the consecutive blocks $B_{i-1} = S[c-b, c+L+d-1]$ and $B_i = S[c, c+L+b+d-1]$;
- if $p \in [c+d+1, c+b-1]$, then w belongs entirely only to the block $B_i = S[c, c+L+b+d-1]$;
- if $p \in [c+b, c+b+d]$, then w belongs to the consecutive blocks $B_i = S[c, c+L+b+d-1]$ and $B_{i+1} = S[c+b, c+L+2b+d-1]$;
- no word of length L can be entirely contained in three or more blocks because $b > d$, and hence no three blocks overlap in a portion of the input string(s) of length at least L.

We define the *merging* of consecutive blocks in the following manner.

Definition 5 (*Block Merging*)**.** Given a window w, let $B_i = S[c, c+b+d+L-1]$ and $B_{i+1} = S[c+b, c+2b+d+L-1]$ be two consecutive blocks. A larger block $B'_{i+1} = S[c, c+2b+d+L-1]$ of size $L+2b+d$ can be obtained by *merging* blocks B_i and B_{i+1}. The definition can be extended in a straightforward way to the merging of k consecutive blocks $B_i, B_{i+1}, \ldots, B_{i+(k-1)}$ of S: merging the k blocks, we obtain an enlarged block $B'_{i+(k-1)} = S[c, c+kb+d+L-1]$ of size $L+kb+d$.

3.2. Description of the algorithm

The algorithm Rime is divided into four main steps, which we now describe.

3.2.1. Step 1: Filtering step

The first step consists simply in applying tuiuiu with a double pass, which includes the optimization step that allows us to discard some false positives due to empty blocks. With respect to the filter introduced in [9,25], in order to collect information which is useful to speed up the next steps, we extend this first phase as follows. For all windows w kept by the filter, information about the non-empty blocks that are friends with w is stored in an array *friendsOfWindow* whose size is the number of possible windows of length L (that is, $n - L + 1$) we recall that n is the total length of the input string(s). The entry *friendsOfWindow[w]* of a window w contains the list of the blocks that are friends of w.

At the end of this step, the portion of the input that is left is the one containing kept windows, that is, those windows w for which the list *friendsOfWindow[w]* has size at least $r - 1$. In this way, a percentage of the initial string(s) is removed, and we are left with actual repeats plus some false positives. Recall that the three possible types of false positive (described in Section 2) are the following.

FP_{block}: due to the choice of checking the filtering condition for windows of size L against blocks of size $L + d + b$.
FP_{cond}: due to fact that the condition the filter checks is a necessary but not sufficient condition.
FP^*: due to the condition being checked between a window and $r - 1$ or more blocks (star-wise) rather than among all such blocks (clique-wise).

Of course, there can be also false positives that are combinations of the three cases above.

3.2.2. Step 2: Semiglobal alignment

In this step, each window kept by the filter after Step 1 is aligned to all the blocks that are its friends. After this step, only windows that have at least $r - 1$ other fragments that are at an edit distance smaller that d are kept. In other words, this step eliminates all FP_{block} and FP_{cond} false positives.

Definition 6 (*Mate Block of a Window*)**.** Given a window w of length L and a block B of length $L + b + d$, we say that B is a *mate* of w if B contains w' such that $d_E(w, w') \leq d$.

During this step, the array *friendsOfWindow* is replaced by a new one, which we call *matesOfWindow*, of the same size and such that *matesOfWindow[w]* contains the list of blocks that are mates of w (that are always a subset of those in *friendsOfWindow[w]*). This is achieved as follows.

For each window w kept after step 1, a semiglobal alignment between w (of length L) and B (of length $L + b + d$) is performed for all blocks B in *friendsOfWindow[w]*. We build a rectangular dynamic programming matrix with w on the rows and B on the columns. The matrix is initialized with zeros on the first row. Indels and mismatches cost 1 and matches cost 0. Since w must be wholly aligned with a substring only of B, we need to check the last row: if there is a value lower or equal to d, then B contains a repeat of w, that is, a substring of length in $[L - d, L + d]$ such that its edit distance from w is at most d; in this case, B is inserted in the list *matesOfWindow[w]*; otherwise, the friendship of B with w was a false positive and B is not inserted in the list *matesOfWindow[w]*. If, for a window w, following the removal of blocks that actually turns *friendsOfWindow[w]* into *matesOfWindow[w]* we obtain that this latter is of size lower than $r - 1$, then w is no longer a window to be kept, and is thus removed.

Each one of these alignments takes time $L(L + b + d)$, and the number of alignments to be performed depends on the dataset and on the efficiency of the filtering phase, which can only be evaluated experimentally (see Section 4). A simple optimization with important practical impact is however possible. This optimization is based on the observation that there exists a relationship between the minimum cost of the alignment of a window w against a block B, and the minimum cost of the alignment of the next window w', that is, the window starting just one position after where w starts if such exists, and the same block B (that is likely to belong to *friendsOfWindow[w']* if it did belong to *friendsOfWindow[w]*). Indeed, in this case we are removing the first row of the alignment between w and B, and adding an extra row at the bottom. If we denote by $dist(win, blo)$ the minimum value at the bottom row of the semiglobal alignment of a window win and a block blo, then we have that

$$dist(w, B) - 1 \leq dist(w', B) \leq dist(w, B) + 1.$$

Therefore, storing for each block B the minimum cost of the alignment with the last aligned window w, it is possible to determine lower and upper bounds of the alignment cost between B and the next window w'. As a result, if $dist(w, B) \in [d, d+1]$, then the alignment between w' and B must be computed, but if $dist(w, B) \leq d-1$ (respectively, $dist(w, B) > d+1$), then we know for free that $dist(w', B) \leq d$ (respectively, $dist(w', B) > d$), and the alignments do not need to be computed.

During this step, new empty blocks can be introduced: a false positive can be detected and discarded, and hence it may turn out that a block belonging to a list *matesOfWindow[w]* for some w is actually empty, that is, no window inside it is kept anymore. For this reason, a strategy of removal of empty blocks is performed also during the alignment step. This has the twofold effect of removing on the fly some FP^* and also of sparing some alignment computations.

At the end of this step, all false positives FP_{cond} and FP_{block} have been removed because now, for all windows w, *matesOfWindow[w]* only stores blocks containing at least one substring x of a length in $[L - d, L + d]$ whose edit distance to w is at most d. Nevertheless, some FP^* possibly remain. These will be addressed in the next step.

3.2.3. Step 3: Clique detection among block mates

At the beginning of this step, we have a set of windows that can either represent a real repeat, or an FP^* false positive. For each such window w, we do know that, in each block belonging to *matesOfWindow[w]*, there is a fragment at edit distance no greater than d to w, but this is not enough to guarantee that w is part of an actual repeat. In order to ensure that, in every pair of blocks $B_i, B_j \in$ *matesOfWindow[w]*, there must be at least two fragments f_i and f_j with $f_i \in B_i$ and $f_j \in B_j$ such that (i) $d_E(f_i, w) \leq d$ (respectively, $d_E(f_j, w) \leq d$), and (ii) $d_E(f_i, f_j) \leq d$. The existence of at least one fragment per block that fulfills condition (i) is guaranteed by the previous steps, but the actual stronger requirement is that the same fragments f_i and f_j that fulfill condition (i) must fulfill condition (ii) as well. Another way to see the problem we are about to address is to represent each window and each fragment f_i as vertices of a graph, and to place an edge between two vertices if these are at edit distance at most d from one another. In this way, the selection made up to this step guarantees that each vertex w is the center of a star-shaped subgraph that has at least $r - 1$ edges, but the actual requirement is for this subgraph to be a clique.

Notice that, should we actually build such a graph, we would have to deal with very noisy data. For example, if a window w has a block B as a mate, then we know that B contains at least one f_i such that $d_E(f_i, w) \leq d$. It is however unlikely that such f_i is unique, because this would mean in particular that $d_E(f_i, w)$ is exactly equal to d. Indeed, if it were that $d_E(f_i, w) = d - \delta$ for some $\delta > 0$, then any other fragment in B at edit distance at most δ from f_i (such as, for example, all fragments obtained from f_i by extending it from one to δ positions at any of its two sides, or deleting from one to δ of its characters, or a combination of the two, etc.) would also correspond to a vertex in the graph and be connected with an edge to w. The result would be a graph with many spurious vertices, and with a locally high density of edges that would in practice heavily slow down a clique detection task because it would get close to the conditions of the worst case of the clique detection problem which is known to be computationally hard.

In order to overcome this drawback, we relax the constraint over the length L and, using the information stored in *matesOfWindow*, build another array of lists, called *matesOfBlock*, where, for each block B, we now store the list of non-empty (possibly overlapping) blocks that are mates of the windows contained in B and kept after the alignment step. Two blocks B_i and B_j are *mate blocks* if there is a fragment $f_i \in B_i$ and a fragment $f_j \in B_j$ such that $d_E(f_i, f_j) \leq d$. The construction of the *matesOfBlock* data structure is performed during the semiglobal alignment step at the same time as the

update of the *matesOfWindow* array. When we find that a window w has at least $r - 1$ non-overlapping blocks that are mates, we detect all the blocks B (there are at most two by Observation 1) that contain w, and add the list of mate blocks stored in *matesOfWindow[w]* to the list *matesOfBlock[B]* for each such B. The *matesOfBlock* data structure is the adjacency list representation of the undirected graph in which we look for maximal cliques of size at least r non-overlapping mate blocks.

In this way, we have reduced our problem to that of finding sets of size at least r of blocks that are pairwise mates: any (L, r, d)-*repeat* would result into such a clique and, moreover, any such clique that is not an (L, r, d)-*repeat* would actually be included in an (L, r, D)-*repeat* with $D > d$. Notice that D is bounded by the fact that the block has size $L + b + d$ and the fact that it is known to contain a fragment at distance d (and not D) from a window of size L. Keeping such (L, r, D)-*repeats* with $D > d$ leads to have theoretically some false positives with respect to the initial target of (L, r, d)-*repeats*. These are still repetitions, but less conserved than (L, r, d)-*repeats*, and hence objects that could still be interesting and that are in general even harder to find, and for these reasons we chose not to spend computational time to take care of discarding them. We must say that in none of our experiments did we actually find (L, r, D)-*repeats* that were not (L, r, d)-*repeats* in the results.

To find maximal cliques in the graph of mate blocks, we use the Bron–Kerbosch [6] algorithm. This is a recursive backtracking procedure that augments a candidate clique by considering one vertex at a time, either adding it to the candidate clique or to a set of excluded vertices that cannot be in the clique but must have some non-neighbor in the eventual clique.

More precisely, we used an optimized version of the Bron–Kerbosch algorithm reported in the same paper [6]. This variant involves the selection of a "pivot" vertex for which two strategies were later investigated in [14]. We tested both selection strategies on several distinct types of biological sequences, and we ended up choosing as pivot the vertex with largest degree because this strategy always outperforms the other one in our experiments. Despite the hardness of clique finding, the characteristics of the graph of mate blocks after all our filtering together with the strategy of the Bron–Kerbosch algorithm, this step turns out – as we shall see in our experiments – to be fast.

3.2.4. Step 4: Removing clique redundancy

After all maximal cliques of the mate blocks graph have been found, there is a final step that refines the results by removing several kinds of possible redundancies in the cliques obtained. We now describe these and the solutions we adopted for their detection and removal. Moreover, we show how some specific redundancies in the set of cliques found during the previous step can actually be due to a wrong setting of the parameters, corresponding for instance to an underestimation of the frequency of a repeat, or of its length, or to an overestimation of the repeat conservation. We show how Rime can detect such events, and then possibly tune the parameters and highlight repeats that are – strictly speaking – false positives with respect to the actual definition of (L, r, d)-*repeat*, but in practice interesting regularities in the input string(s).

Merging the blocks. It is possible that two different cliques actually represent a same repeat. Indeed, consider a pair of consecutive entries i and $i + 1$ in *matesOfBlock* corresponding to two consecutive blocks B_i and B_{i+1} that share a window w kept after the semiglobal alignment step because it has at least $r - 1$ non-overlapping mate blocks. If w is not an FP^*, the Bron–Kerbosch algorithm would find two cliques: $C = B_i \cup$ *matesOfWindow[w]* and $C' = B_{i+1} \cup$ *matesOfWindow[w]*. Aligning the blocks of both cliques C and C', the same repeat is found, as the only two blocks in which C and C' differ contain the same occurrence w of the repeat.

This kind of redundancy in the output can be avoided by storing in *matesOfBlock[i]* and *matesOfBlock[i+1]*, when they are created in place of *matesOfWindow[w]*; also the blocks B_{i+1} (respectively B_i). In this way, the Bron–Kerbosch algorithm would find only the clique $C = \{B_i, B_{i+1}\} \cup$ *matesOfWindow[w]*.

On the other hand, the same type of redundancy now occurs within a single clique because B_i and B_{i+1} represent the same occurrence of a repeat. In particular, it can be the case now that the size of the clique is erroneously judged large enough (i.e., with at least r blocks), while the real number of occurrences of the repeat is smaller. In order to remove this type of redundancy inside a clique, we merge the consecutive blocks composing it. A clique $C = \{B_i, B_{i+1}\} \cup \mathcal{B}$ (where $\mathcal{B}$ is a set of blocks) thus becomes $C' = B'_{i+1} \cup \mathcal{B}$. This merging inside cliques is performed when a new block is added to a candidate clique. Therefore, the frequency check (i.e., the check whether the size of the clique is at least r) is done after the merging operations. An example of the merging of consecutive or overlapping blocks is shown in Fig. 1. Observe that the block merging operation is applied also to overlapping blocks that are present inside a clique because they represent overlapping occurrences of a same repeat (which we are not interested in detecting), while no two non-overlapping occurrences of a same repeat can be included in two consecutive blocks unless $b > L$.

A clique may include more than two consecutive or overlapping blocks if they contain overlapping occurrences of a same repeat. In this case, only one block that is the union of all consecutive and overlapping blocks is returned as part of the clique. In the alignment, we therefore see only one long occurrence. When multiple strings are given as input to the algorithm, and we search for repeats occurring in at least r strings, this procedure requires an exhaustive enumeration of the cliques representing all the existing repeats because, even merging consecutive and overlapping blocks, we keep at least one block for each string in which at least one occurrence of the repeat represented by the clique exists.

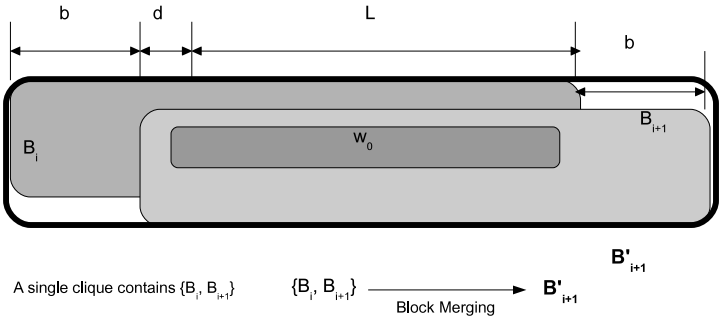

Fig. 1. *Block Merging* operation: blocks B_i,B_{i+1} belong to a same clique representing the same occurrence of a repeat, and thus are merged into a unique block inside the candidate clique.

When there is only one string given as input, and we look for repeats having at least r non-overlapping occurrences in the string, on the contrary, we may miss some repeats. Indeed, the described procedure may merge all blocks that contain an occurrence of a repeat if they all are consecutive, but some non-overlapping occurrences of the repeat may be present in these blocks and hence are missed. In order to avoid this problem, in the version of the algorithm for a single input string, the operation of merging is performed for at most $\left\lfloor \frac{L-(b+d)-1}{b} \right\rfloor$ consecutive blocks. In this way, we do not miss any non-overlapping occurrence of a repeat because this limited number of windows spans over a portion of the input in which there is not enough room for two non-overlapping occurrences of a repeat.

Enlarging the blocks. We have seen how to remove redundancy within a clique. We now show another kind of redundancy that involves two or more cliques that, indeed, turn out to actually represent different parts of a same repeat. Consider, for example, the following set of cliques with $r = 2$:

$$\begin{aligned} C &= B_i, B_j \\ C^1 &= B_{i+1}, B_{j+1} \\ C^2 &= B_{i+2}, B_{j+2} \\ &\vdots \\ C^k &= B_{i+k}, B_{j+k} \end{aligned}$$

In this case, it is plausible to think that in the input strings there exists a repeat of size greater than L whose occurrences are the concatenation of the occurrences of shorter $k + 1$ overlapping repeats, each one being represented by one of the cliques $C, C^1, C'^2, \ldots, C^k$. This means that the user underestimated the length L of the repeats to be sought. This produces an output that is difficult to read and manage because there is a redundant number of cliques for the same repeat. To address this problem, in this case, we merge consecutive blocks contained in the consecutive cliques, and return only the clique $C = B'_{i+n}, B'_{j+n}, B'_{k+n}, B'_{t+n}$ representing the longer repeat. We denote the merging of consecutive blocks between different cliques by *Block Enlarging*. An example is shown in Fig. 2.

The two situations – having consecutive and overlapping blocks inside the same cliques and in different cliques – may happen simultaneously.

In addition, it may happen also that only some occurrences of a longer repeat are the concatenation of occurrences of shorter repeats. In this case, we could report only the clique composed of blocks obtained by merging consecutive blocks in different cliques. Then, looking at the alignment of blocks in C, it will be clear that there exists a short repeat that can be further extended, possibly losing some occurrences. On the contrary, the merging of consecutive blocks contained in two different cliques is not performed if there exist at least two blocks that are not consecutive or overlapping in the two cliques.

3.3. Tuning the parameters

When we search for cliques of maximal size (and not just of size at least r), we may find one clique C of size $R > r$. In that case, we only output that repeat and, in particular, we do not output any clique among those included in C. Specifically, we avoid returning repeats that occur from r to $R - 1$ times, and hence formally fail to output a repeat that satisfies the initial requirements. This is made to avoid a redundant output, and actually suggests to the user that, at least for that repeat, the frequency, and hence the parameter r, had been underestimated. This is the first kind of parameter tuning performed by Rime.

The second type of parameter tuning takes place when we perform block enlarging: we actually find repeats of size larger than L and, consequently, with more than d pairwise differences. These can be obtained by putting together all the repeats of consecutive blocks (that overlap), like u and v in Fig. 3, where an example of the enlarging of two blocks is shown.

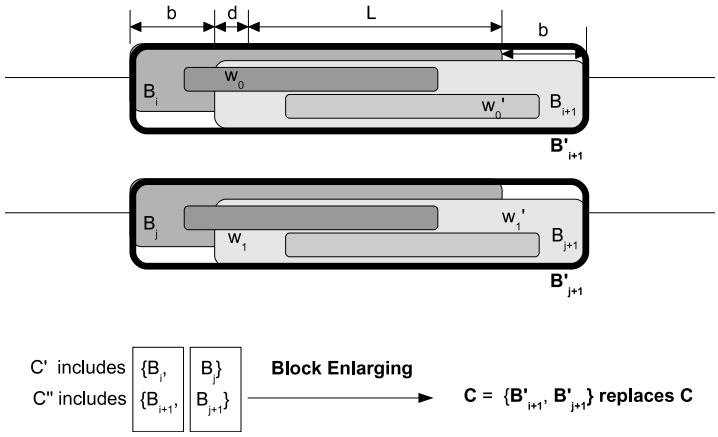

Fig. 2. *Block Enlarging* operation: windows w_0 and w_1 are two occurrences of a same repeat contained respectively in blocks B_i and B_j. The blocks B_{i+1} and B_{j+1} contain windows w_0' and w_1' which overlap w_0 and w_1 respectively, and are occurrences of another repeat. The Bron–Kerbosch algorithm finds two cliques C' and C'' composed of consecutive blocks; but actually, in the strings, there exists a longer repeat whose occurrences are the concatenation of w_0 and w_1, and that of w_0' and w_1'. The *Block Enlarging* operation consists in merging consecutive blocks between different cliques.

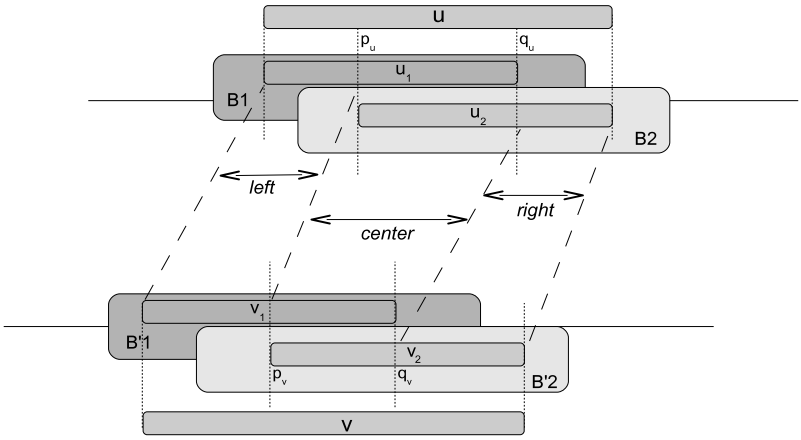

Fig. 3. *Block Enlarging* for two blocks: distances of enlarged repetitions.

The following result characterizes the kind of repeats we find in that case.

Proposition 1. *Let B, B' be a pair of enlarged blocks of k consecutive blocks, and let rep be the consequently enlarged repeat. The enlarged blocks contain occurrences of the repeat rep having length at most $L + kb + d$ with at most kd pairwise differences.*

Proof. Let B and B' be any pair of enlarged blocks obtained from, respectively, the consecutive blocks $B_1, B_2, \ldots, B_k$ and $B_1', B_2', \ldots, B_k'$, and let the two enlarged repeats u (in B) and v (in B') be obtained from, respectively, $u_1, \ldots, u_k$ and $v_1, \ldots, v_k$ such that, for $1 \le i \le k$, we have that u_i (respectively, v_i) is the occurrence included in B_i (respectively, in B_i') of the ith out of k (L, r, d)-*repeats*. An enlarged block built from k consecutive blocks has size $L + kb + d$, and thus this is the upper bound on the size of any occurrence of repeat it may contain.

We now show that $d_E(u, v) \le kd$. Let us first prove the result for $k = 2$, and then we show how to extend the proof to any value of k. The situation with $k = 2$ is depicted in Fig. 3. Let p_u (respectively, p_v) be the position of u_1 (respectively, v_1) where this starts overlapping u_2 (respectively, v_2), and let q_u (respectively, q_v) be the position of u_2 (respectively, v_2) where this stops overlapping u_1 (respectively, v_1). These are shown in Fig. 3. Notice that p_u and p_v are not necessarily equal, and the same holds for q_u and q_v. Notice also that $u_1[p_u, |u_1|]$ is identical to $u_2[1, q_u - 1]$ (respectively, $v_1[p_v, |v_1|] = v_2[1, q_v - 1]$) because they represent the same fragment of the input string. Moreover, let u (respectively, v) be the occurrence of the enlarged repeat: u (and similarly v, which we do not detail) is composed of the prefix $u_1[1, p_u - 1]$ of u_1 concatenated to $u_1[p_u, |u_1|] = u_2[1, q_u - 1]$ and to the suffix $u_2[q_u, |u_2|]$ of u_2. By hypothesis, we have that $d_E(u_1, v_1) = d_1 \le d$ and $d_E(u_2, v_2) = d_2 \le d$. Let us consider the optimal alignment of u_1 and v_1: this can be seen as the concatenation of that involving the prefix $u_1[1, p_u - 1]$ of u_1 (corresponding to the area indicated as 'left' in Fig. 3), and that involving the remaining $u_1[p_u, |u_1|]$ (corresponding to the area indicated as 'center'); let us denote by d_{left} the number of edit operations of the first portion of the alignment, and by d_{center_1} the number of edit operations of the second portion. We have that $d_1 = d_{left} + d_{center_1} \le d$. Similarly, taking into account the best alignment of u_2 and v_2, we can state that

Table 1
Performances of the different steps of RIME to find (L, r, d)-*repeats* in the Sunflower *bacKnapp* dataset (107 161 bases).

L	d	Filter		Semiglobal align		Clique detection		Total
		Time (s)	Sel	Time (s)	Sel	Time	#cliques	Time (s)
200	20	0.50	12.47%	5.13	10.32%	0.00	8	5.63
300	30	0.49	12.12%	10.85	9.85%	0.01	5	11.34

$d_2 = d_{center_2} + d_{right} \leq d$, where d_{center_2} is the number of edit operations between $u_2[1, q_u - 1]$ and $v_2[1, q_v - 1]$, while d_{right} is the difference between what remains of both (in the area denoted by 'right' in Fig. 3). Observe that, given that $u_1[p_u, |u_1|] = u_2[1, q_u - 1]$ and $v_1[p_v, |v_1|] = v_2[1, q_v - 1]$, then $d_{center_1} = d_{center2}$, and hence we call them both just d_{center}. We can build an alignment of u and v simply concatenating the left portion of the alignment of u_1 and v_1, the alignment in the center for the fragments where u_1 and u_2 overlap against the overlap of v_1 and v_2, and the right portion of the alignment between u_2 and v_2. Therefore, we have an alignment of u and v with only $d_{left} + d_{center} + d_{right}$ edit operations, and hence $d_E(u, v) \leq d_{left} + d_{center} + d_{right} \leq d_1 + d_2 \leq 2d$.

The case of $k > 2$ is a straightforward extension of the one for $k = 2$. What changes is that there are $k - 1$ areas $center_1, \ldots, center_{k-1}$, and k alignments between u_i and v_i for each $i = 1, \ldots, k$. Similarly, the numbers of edit operations in the internal portions of the alignments all collapse to be equal within the same column '$center_j$' for $j = 1, \ldots, k - 1$. Therefore, the outcome is that $d_E(u, v) \leq d_{left} + d_{center_1} + \cdots + d_{center_{k-1}} + d_{right} \leq d_1 + \cdots + d_k \leq kd$. This concludes the proof. □

It is worth observing that the bound on the distance in Proposition 1 above is tight only in the case in which each d_i is equal to d, and all d_{center} are equal to 0. Although theoretically possible, this is quite unlikely: it would mean that there are as many as d total differences between u_i and v_i for each $1 \leq i \leq k$, and that, given that it must be that all d_{center} are 0, they are all in the 'left' region between u_1 and v_1 and in the 'right' region between u_k and v_k: this is a very special case that can only hold only for $k = 2$.

4. Experiments and discussion

4.1. Validation of RIME

We now show the results of an extensive set of tests performed to validate RIME for finding (L, r, d)-*repeats* in biological datasets. The datasets selected were as follows. 1: a set of BAC sequences of the Sunflower (*bacKnapp* dataset), 2: the whole genomes of three strains of *Saccharomyces cerevisiae* [18], and 3: a dataset of sequences obtained from [3] and which represent the ortholog regions of the cystic fibrosis transmembrane conductance regulator (CFTR) gene for five different organisms. Performances of the different steps of RIME are evaluated in terms of running time and of selectiveness relatively to the amount of data left after the first and the second steps (that is, the filtering and then the semiglobal alignment that removes FP_{cond} and FP_{block}). The selectiveness of these two steps is defined as the ratio between the number of non-removed overlapping substrings of length L and the total number of overlapping substrings of length L present in the input sequences. Formally, the selectiveness of both steps (1) and (2) of RIME is given by $sel = \frac{\text{number of words of length L kept by RIME step}}{\text{number of words of length L in the input sequences}}$. Obviously, given that both steps are lossless, the smaller the selection, the better. On the other hand, for steps (3) and (4) (that is, clique detection and clique redundancy removal, respectively), we report the number of cliques that are found.

A possible application of (L, r, d)-*repeats* is to the detection of LTR sequences (LTR is the acronym for Long Terminal Repeats, which are sequences of about 300 bp length repeated at both ends of a transposable element). As a first experiment, we applied RIME to four different datasets composed of a single BAC sequence of the Sunflower, using length parameters in agreement with the expected structure of LTRs ($L = 200, 300$, with $d = 20, 30$, respectively).

Tables 1 and 2 report the results of RIME for one of these four datasets denoted by *bacKnapp* and containing 107 161 bases, using respectively $r = 2$ and $r = 3$. The results for the other three datasets were essentially equivalent to those we report here (data not shown).

Analyzing in detail the performance of each single step of RIME, we observe that, as expected, the most time-consuming part is the semiglobal alignment between windows and friend blocks (except for the special case of the last line of Table 2, which we specifically comment on later). However, we are able to perform the alignment task in reasonable time for all parameters (and this holds for all the four datasets) because the preprocessing filtering step sensibly reduces the input size.

The other step with a high theoretical computational complexity is the clique detection performed on the graph of mate blocks. However, we observe that, even though the Bron–Kerbosch algorithm applied to an n-vertex graph has a time complexity exponential in n, the clique detection step is very fast in all tests, and even more when $r = 3$ instead of 2, because it is performed on small graphs of mate blocks thanks to the filtering of the input sequences and the small number of false positives that remain after the semiglobal alignment step. As concerns the number of detected cliques, we can see that our strategy of compression of the output allows us to obtain smaller one. Indeed, RIME returns very few repeats, especially when $r = 3$. Finally, the last two lines of Table 2 report tests in which the allowed edit distance is pushed quite far (45 edit operations allowed in a 300 base long repeat, meaning 15% of the repeat length). No new result appears in this case, but we can see that the time performances of RIME are good, even if the filter helps much less and takes more time.

Table 2
Performances of the different phases of RIME to find (L, r, d)-*repeats* on the Sunflower *bacKnapp* dataset (107 161 bases), with $r = 3$.

L	d	Filter		Semiglobal align		Clique detection		Total
		Time (s)	Sel	Time (s)	Sel	Time	#cliques	Time (s)
200	20	0.44	3.32%	3.38	1.10%	0.00	3	3.82
300	30	0.46	3.42%	7.36	0.98%	0.00	2	7.82
200	25	0.59	5.66%	4.24	2.57%	0.00	3	4.83
300	45	178.25	41.70%	35.59	3.15%	0.00	2	213.84

Table 3
Performances of the different phases of RIME to find (L, r, d)-*repeats* on the *s288c+w303* dataset (26 392 324 bases) of the *S. cerevisiae*, with $r = 3$.

L	d	Filter		Semiglobal align		Clique detection		Total
		Time (s)	Sel	Time (s)	Sel	Time	#cliques	Time (s)
200	20	29.44	0.17%	744.48	0.09%	6.30	24	780.22
300	30	31.68	0.16%	1473.65	0.07%	2.13	13	1507.46
5000	500	9.00	0	–	–	–	–	9.00

In addition, in order to validate our results, we compared the repeats found by RIME in the Sunflower with the ones found by the signature-based repeat finding method LTR_FINDER [33]. Given that no annotation is available yet, the output of such a tool is the only result we can compare ours to. We observed that all the repeats identified by the other method are found also by RIME. The latter however returns also further repeats which are not identified by the former. These results suggest that RIME can provide a fast solution to the problem of finding long repeats representing LTRs.

We performed experiments on the dataset *s288c+w303* composed of the whole genomes (16 chromosomes each) of three different strains of *S. cerevisiae*: RefSeq (that is fully annotated in the *Saccharomyces* Genome Database), S288c and W303, for a total of 26 392 324 bases. The dataset was preprocessed by the REGENDER algorithm [4] (the reported size is the one after REGENDER is applied) in order to remove the resident genome (i.e., the total immotile DNA), which is equal among all the strains and does not contain mobile elements like transposons. The goal of applying RIME to this dataset is to detect transposable elements and LTRs that are shared by the three strains, and that could not be detected by means of a traditional global alignment because, in general, being part of the most mobile DNA, they have lost their co-linearity.

Table 3 reports the results of the tests performed to find (L, r, d)-*repeats* characterized by the following parameters: $r = 3$, $L = 200, 300, 5000$ with $d = 20, 30, 500$, respectively, in the *s288c+w303* dataset. We chose these parameters based on the peculiar structure of the transposons that can be found from the annotation of RefSeq provided in the *Saccharomyces* Genome Database (available at http://www.yeastgenome.org): they are long, between 5000 and 6000 bases, and are delimited by two LTRs of 200–300 bases.

Basically, all the observations we made for the Sunflower data set hold here as well, including the fact that our algorithm is a good candidate to detect LTRs. For this dataset indeed an annotation is available, and hence in this case we could validate our results. In particular, we checked whether the repeats found by RIME in these strains of *S. cerevisiae* correspond to real LTRs whose annotation is available in the *Saccharomyces* Genome Database (http://www.yeastgenome.org). We found that the repeats identified using parameters $L = 300$, $d = 30$, $r = 2$ actually either correspond to real LTRs, or are part of retrotransposons, or else match with the sequences of putative proteins of unknown function. For example, the blocks composing a detected clique contain occurrences of the following annotated LTRs: YCLWdelta3 and YCLWdelta5 in chromosome III, YDRWdelta19 and YDRWdelta28 in chromosome IV, and YLRWdelta14 and YLRWdelta23 in chromosome XII. For longer repeated sequences such as transposons and retrotransposons, nothing is selected, as expected, probably because the maximum of 10% differences allowed is not the right framework to capture the divergence among transposons.

4.2. Comparison with other algorithms

To the best of our knowledge, RIME is the first *ab initio* non-heuristic algorithm that can deal with repeats occurring in possibly more than two sequences, that have length of hundreds or thousands of bases, and whose occurrences may differ in 10% or even more of their positions in terms of substitutions and indels. For this reason, we cannot compare RIME with other methods solving the same problem. In this section, we report instead the results of experiments performed to compare RIME with existing methods for local similarity search. In particular, as the major strength of RIME is its capacity to identify repeats in more than two input sequences, we concentrated our attention on existing methods for multiple local sequence alignment. It is important to observe, however, that the output provided by RIME and the one provided by multiple local alignment approaches are different because the tasks addressed by the two kinds of method are not the same. Indeed, RIME returns distinct alignments for each found repeat and its occurrences, while the output of multiple local alignment methods is the alignment of whole input sequences in which we can identify local similarity areas (the repeats) by looking at the alignment.

We compared RIME to some of the most popular multiple local alignment methods on the CFTR dataset [3] composed of five ortholog regions of the cystic fibrosis transmembrane conductance regulator gene in chicken, cow, human, mouse

Table 4
Performances of the different steps of RIME to find (L, r, d)-*repeats* in the CFTR dataset (5 518 041 bases), with $L = 100$ and $r = 5$.

d	Filter		Semiglobal align		Clique detection		Total
	Time (s)	Sel	Time (s)	Sel	Time	#cliques	Time (s)
7	64.20	0.05%	56.56	0	–	–	120.76
12	1017.51	0.01%	0.88	0	–	–	1018.39
14	3772.65	0.02%	1.41	0.001%	0.00	1	3774.06
15	7128.19	0.65%	740.01	0.003%	0.01	1	7868.21

Table 5
Results of several multiple local sequence alignment tools on the CFTR dataset.

Tool	Class	Result
MSA [17]	Exact	Manages sequences of a total length of at most 12 500 characters
ClustalW [31]	Progressive	Runs for more than 38 h
TCoffee [21]	Progressive	Runs out of memory
Kalign [16,15]	Progressive	Runs for more than 28 h
DiAlign [30]	Iterative	Runs out of memory
MUSCLE [8]	Iterative	Runs out of memory

and tetra. This is the smallest dataset (5.5 Mbases) composed of more than two sequences that we have studied in our work. Experiments were run on an Intel®Quad-core Xeon®E5405/2 GHz with 10 GB of RAM. Table 4 reports the results of experiments performed on the CFTR dataset. We used the same parameter settings as for the experiments reported in the previous section, except that here we set $r = 5$. The whole set of parameters is thus $L = 100$, $r = 5$ and $d = 7, 12, 14, 15$. We can see that, for low values of d, no repeat is detected, while, for larger values, there is a repeat that our algorithm is fast to find, despite the fact that its occurrences pairwise show 15% differences. We have tried to search for other methods able to find the same results with which we could compare the performances of RIME. Table 5 summarizes the results of the comparison. As we can see, none of the tested methods was able to manage, in reasonable time and without huge memory usage, inputs as large as the one provided by the sequences in the CFTR dataset. In contrast, as shown in Table 4, RIME ends its computation in reasonable time on this dataset with these parameters.

5. Conclusion and perspectives

The problem of finding long repeats is computationally challenging when a non-negligible number of insertions, deletions and substitutions are allowed among the occurrences of the repeats. An exhaustive discovery of such repeats might indeed be unfeasible for many instances. RIME is, to the best of our knowledge, the first algorithm that can deal with repeats occurring possibly several times, having a length up to a few thousands of bases, and whose occurrences may differ in 10% or more of their positions among substitutions and indels. This is achieved by using a filter as a preprocessing step, and using the information gathered during the filtering phase in order to then speed up a dynamic-programming-based alignment step performed to infer the repeats. Although in theory the current version of RIME might return some false positives due to the introduction of a localized heuristic step in the method, we never observed such false positives in practice. A possible future work will consist in clearly evaluating this false positive rate and in finding a new way for addressing the problem. We also intend to investigate further the application of RIME for finding mobile elements in genomes. Following the studies of [4,20], the tool could be applied to whole datasets of many strains of the same genome that have been preprocessed by removing the resident genome. Finally, we can use RIME also to detect repeated regions of genomes by examining raw sequencing data only and detecting fragments that are repeated sensibly more than the sequencing coverage; in this way we could extend our [26] approach to the detection of structural variants larger than point mutations and also resolve loops in bubble detection on de Bruijn graphs [5].

Acknowledgments

The research leading to these results has received funding from the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013)/ERC grant agreement n [247073]10, the French project ANR MIRI BLAN08-1335497, and the Italian MIUR PRIN project ALGODEEP prot. 2008TFBWL4.

References

[1] G. Achaz, F. Boyer, E. Rocha, A. Viari, E. Coissac, Repseek, A tool to retrieve approximate repeats from large DNA sequences, Bioinformatics 23 (1) (2007) 119–121.
[2] A. Apostolico, M. Comin, L. Parida, VARUN: Discovering extensible motifs under saturation constraints, IEEE/ACM Trans. Comput. Biology Bioinform. 7 (4) (2010) 752–762.
[3] M. Brudno, C.B. Do, G.M. Cooper, M.F. Kim, E. Davydov, E.D. Green, A. Sidow, S. Batzoglou, LAGAN and Multi-LAGAN: efficient tools for large-scale multiple alignment of genomic DNA, Genome Res. 13 (2003) 721–731.

[4] G. Battaglia, R. Grossi, N. Pisanti, R. Marangoni, G. Menconi, Inferring mobile elements in S.cerevisiae strains, in: Proceedings of International Conference on Bioinformatics Models, Methods and Algorithms, BIOINFORMATICS, 2011, pp. 131–136.
[5] E. Birmelé, P. Crescenzi, R.A. Ferreira, R. Grossi, V. Lacroix, A. Marino, N. Pisanti, G.A.T. Sacomoto, M.-F. Sagot, Efficient bubble enumeration in directed graphs, in: Proceedings of 19th Symposium on String Processing and Information Retrieval, SPIRE, 2012, pp. 118–129.
[6] C. Bron, J. Kerbosch, Algorithm 457: finding all cliques of an undirected graph, Commun. ACM 16 (9) (1973) 575–577.
[7] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, M. Vingron, *q*-gram based database searching using a suffix array (QUASAR), in: ACM Conference on Research in Computational Molecular Biology, RECOMB 1999, 1999, pp. 77–83.
[8] R.C. Edgar, Muscle: multiple sequence alignment with high accuracy and high throughput, Nucleic Acids Res. 32 (2004) 1792–1797.
[9] M. Federico, P. Peterlongo, N. Pisanti, An optimized filter for finding multiple repeats in DNA sequences, in: Proceedings of the 8th IEEE International Conference on Computer Systems and Applications, AICCSA 2010, IEEE Computer Society Press, 2010, pp. 1–8.
[10] M. Federico, P. Peterlongo, N. Pisanti, M.-F. Sagot, Finding long and multiple repetitions with edit distance, in: Proceedings of the Prague Stringology Conference, 2011.
[11] R. Grossi, A. Pietracaprina, N. Pisanti, G. Pucci, E. Upfal, F. Vandin, MADMX: A strategy for maximal dense motif extraction, J. Comput. Biol. 18 (4) (2011) 535–545.
[12] C.S. Iliopoulos, J.A.M. McHugh, P. Peterlongo, N. Pisanti, W. Rytter, M.-F. Sagot, A first approach to finding common motifs with gaps, Internat. J. Found Comput. Sci. 16 (6) (2005) 1145–1154.
[13] J. Jones, M. Gellert, The taming of a transposon: V(D)J recombination and the immune system, Immunol. Rev. 200 (1) (2004) 233–248.
[14] I. Koch, Enumerating all connected maximal common subgraphs in two graphs, Theoret. Comput. Sci. 250 (2001) 1–30.
[15] T. Lassmann, O. Frings, E. Sonnhammer, Kalign2: high-performance multiple alignment of protein and nucleotide sequences allowing external features, Nucleic. Acids. Res. 37 (3) (2009) 858–865.
[16] T. Lassmann, E.L. Sonnhammer, Kalign — an accurate and fast multiple sequence alignment algorithm, BMC Bioinformatics 6 (2005) 1–20.
[17] D.J. Lipman, S.F. Altschul, J.D. Kececioglu, A tool for multiple sequence alignment, Proc. Nat. Acad. Sci. 86 (1989) 4412–4415.
[18] G. Liti, D.M. Carter, A.M. Moses, et al., Population genomics of domestic and wild yeast, Nature 458 (2009) 337–341.
[19] L. Marsan, M.-F. Sagot, Algorithms for extracting structured motifs using a suffix tree with an application to promoter and regulatory site consensus identification, J. Comput. Biol. 7 (3–4) (2000) 345–362.
[20] G. Menconi, G. Battaglia, R. Grossi, N. Pisanti, R. Marangoni, A Taste of Yeast Mobilomics, in: Proceedings of International Conference on Bioinformatics Models, Methods and Algorithms, BIOINFORMATICS, 2012, pp. 271–274.
[21] C. Notredame, D.G. Higgins, J. Heringa, T-Coffee: a novel method for fast and accurate multiple sequence alignment, J. Mol. Biol. 302 (2000) 205–217.
[22] V. Pereira, D. Enard, A. Eyre-Walker, The effect of transposable element insertions on gene expression evolution in rodents, PLoS one 4 (2) (2009) e4321.
[23] P. Peterlongo, N. Pisanti, F. Boyer, A.P. do Lago, M.-F. Sagot, Lossless filter for multiple repetitions with hamming distance, J. Discrete Algorithms 6 (3) (2008) 497–509.
[24] P. Peterlongo, N. Pisanti, F. Boyer, M.-F. Sagot, Lossless filter for finding long multiple approximate repetitions using a new data structure, the bi-factor array, in: Proceedings of 12th Symposium on String Processing and Information Retrieval, SPIRE, 2005, pp. 179–190.
[25] P. Peterlongo, G.T. Sacomoto, A.P. do Lago, N. Pisanti, M.-F. Sagot, Lossless filter for multiple repeats with bounded edit distance, Algorithms for Mol. Biol. 4 (3) (2009) 1–20.
[26] P. Peterlongo, N. Schnel, N. Pisanti, M.-F. Sagot, V. Lacroix, Identifying SNPs without a reference genome by comparing raw reads, in: Proceedings of 17th Symposium on String Processing and Information Retrieval, SPIRE, 2010, pp. 147–158.
[27] N. Pisanti, M. Giraud, P. Peterlongo, Filters and seeds approaches for fast homology searches in large datasets, in: M. Elloumi, A.Y. Zomaya (Eds.), Algorithms in Computational Molecular Biology, John Wiley & Sons, 2010, pp. 299–320 (Chapter 15).
[28] K. Rasmussen, J. Stoye, E. Myers, Efficient q-gram filters for finding all epsilon-matches over a given length, J. Comput. Biol. 13 (2) (2006) 296–308.
[29] S.E. Rombo, Extracting string motif bases for quorum higher than two, Theoret. Comput. Sci. 460 (2012) 94–103.
[30] A. Subramanian, M. Kauffmann, B. Morgenstern, DIALIGN-TX: greedy and progressive approaches for segment-based multiple sequence alignment, Algorithms for Mol. Biol. 3 (2008). Art. 6.
[31] J.D. Thompson, D.G. Higgins, T.J. Gibson, Clustal W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice, Nucleic Acids Res. 22 (1994) 4673–4680.
[32] T. Treangen, S. Salzberg, Repetitive DNA and next-generation sequencing: computational challenges and solutions, Nature Rev. Genetics 13 (2012) 36–46.
[33] Z. Xu, H. Wang, LTR_FINDER: an efficient tool for the prediction of full-length LTR retrotransposons, Nucleic Acids Res. 35 (2007) W265–W268.