RISOTTO: Fast extraction of motifs with mismatches

Nadia Pisanti[◇‡], Alexandra M. Carvalho[°*], Laurent Marsan[∓] and Marie-France Sagot[◇]
[°] INESC-ID, Lisbon
[◇] INRIA Rhône-Alpes, France
[‡] Dipartimento di Informatica, Università di Pisa, Italy
[∓] Institut Gaspard-Monge, Université de Marne-la-Vallée, France

Abstract

We present in this paper an exact algorithm for motif extraction. Efficiency is achieved by means of an improvement in the algorithm and data structures that applies to the whole class of motif inference algorithms based on suffix trees. An average case complexity analysis shows a gain over the best known exact algorithm for motif extraction, when applied to extract long motifs. A full implementation was developed and made available online. Experimental results show that the proposed algorithm is more than two times faster than the best known exact algorithm for motif extraction, confirming in this way the theoretical results obtained.

Keywords: Bioinformatics, Motif extraction algorithm, Complexity, Maximal extensibility.

Availability: `http://algos.inesc-id.pt/~asmc/software/riso.html`

1 Introduction

Patterns appearing repeated either inside a same string or over a set of strings are important objects to identify. Such repeated patterns are called *motifs* and their identification is called *motif inference* or *motif extraction*.

The area has many potential applications, namely to data compression [4], natural languages, databases, basically, any activity or research requiring text mining [9]. The field of application that concerns us is molecular biology. The motifs in this case may correspond to functional elements in DNA, RNA or protein molecules, or to whole genes whose sequences are strongly similar. In biological applications, it is mandatory to allow for some mismatches between different occurrences of the same motif. In fact point mutations might have taken place, as well as errors in the sequencing procedure, so that molecules that have the same or related function(s), have no longer identical sequences. This is what makes the problem difficult from the computational point of view.

In this paper we propose an exact algorithm for the extraction of motifs with mismatches. In particular, we consider *single* and *structured motifs*, which are motifs composed of several disjoint single motifs placed at given distances from each other. The extraction of structured motifs appears particularly interesting because of its application to the detection of binding

[*]Partially supported by FCT grant SFRH/BD/18660/2004 and FCT Project FEDER POSI/SRI/47778/2002 BioGrid.

sites that respect a distance constraint (see for instance [7] for a biological motivation for structured motifs). Given a text s, the problem is to find repeated patterns in s according to some parameters that specify the frequency and the structure required for the motifs. In molecular biology, the text is in general a set of DNA sequence.

Several exact [10, 3], heuristic [7, 8] and probabilistic [16, 15] algorithms for extracting structured motifs exist. Up to date, the best known exact algorithms for the extraction of single [13] and structured [3] motifs perform well when searching for short motifs. In this paper, we propose an improvement to such algorithms in order to deal with long motifs. The problem of extracting long motifs was first adressed by Pevzner and Sze [11]. They considered a precise version of the motif discovery problem: find all single motifs of length 15 with at most 4 mismatches in 20 sequences of size 600. In consequence several algorithms appeared [11, 2, 5, 12]. A general set for this problem deserves attention from the algorithmic point of view because its computational complexity is in the worst case exponential with respect to the number e of mismatches allowed among different occurrences of the same motif. The reason is that, to identify motifs of the required length, there can be an explosion of the number of candidates of intermediate length whose extension has to be attempted. This imposes in practice a limit to the length of the motifs themselves, as in many applications the value of e depends on this length. The improvement introduced in this paper acts exactly in these cases, and hence applies to relatively long motifs, being a way to increase the length of motifs that are detectable in practice.

2 Single motif extraction

A *single motif* is a word over an alphabet Σ. Given an error rate e, a motif is said to *e-occur* in a sequence if it occurs with at most e letters substitution. The *single motif extraction problem* takes as input N sequences, a quorum $q \leq N$, a maximal number e of mismatches allowed, and a minimal and maximal length for the motifs, k_{min} and k_{max}, respectively. The problem consists in determining all motifs that e-occur in at least q input sequences. Such motifs are called *valid*. An efficient exact algorithm for the extraction of single motifs with mismatches has been introduced in [13] and is based on a suffix tree. In a few words, motifs are considered in lexicographical order starting from the empty word, and they are extended to the right as long as the quorum is satisfied until either a valid motif of maximal length is found (if the k_{max} length is reached), or the quorum is no longer satisfied. In both cases, a new motif is attempted. At each step, all nodes spelling e-occurrences of the current motif are taken into account. More formally, the algorithm presented in [13] we refer to is sketched in Algorithm 1, where motif m is the one whose extension is being tried. At the beginning **ExtractSingleMotif**

Algorithm 1 Single motif extraction

ExtractSingleMotif(motif m)
1. **for all** $\alpha \in \Sigma$ **do**
2.   **if** $m\alpha$ is valid **then**
3.   **if** $|m\alpha| \geq k_{min}$ **then** spell out the valid motif $m\alpha$
4.   **if** $|m\alpha| < k_{max}$ **then** **ExtractSingleMotif**($m\alpha$)

is called on the empty word. The algorithm recursively calls itself for longer motifs built by adding letters (step 4), and considers new ones (step 1) when the extension fails (step 2).

A valid motif is spelled out whenever a motif whose length lies within the required minimal and maximal length is being considered (step 3). The order in which motifs are generated corresponds to a depth-first visit of a complete trie $\mathcal{M}$ of all words of length k_{max} on the alphabet Σ. We refer to $\mathcal{M}$ as the *motif tree*. In fact, the algorithm does not need to allocate the motif tree. The only memory requirement is for the suffix tree $\mathcal{T}$. Assuming that the required length of the motif is k (that is $k_{min} = k_{max} = k$), and that at most e mismatches are allowed, the algorithm has worst case time complexity in $O(Nn_k\nu(e,k))$, where n_k is the number of tree nodes at depth k, and $\nu(e,k)$ is the number of words of length k that differ in at most e letters from a word m of length k. This value does not depend on m, and it holds that $\nu(e,k) \leq k^e|\Sigma|^e$. This upper bound is in practice not tight. Nevertheless, no better bound can be given and therefore the time complexity is linear in the input size, but possibly exponential in the number e of mismatches. Since reasonable values for e are proportional to the value of k, this actually places a practical bound on the length required for the motifs. The goal of this paper is to move this bound. Finally, the space complexity is $O(Nn_k)$.

2.1 Using maximal extensibility of factors

The modification we suggest consists in storing information concerning maximal extensibility in order to avoid trying to extend hopeless motifs. For instance (see Figure 1), assume that in our virtual depth-first visit of the motif tree, we have found out that motif m can be further extended without losing the quorum up to a length of $MaxExt(m)$ only, the latter representing its maximal extensibility. If later on, we are processing a motif m' that has m as a suffix, then the $MaxExt(m)$ information could be useful, as it applies to m' as well because m' can also be extended with at most $MaxExt(m)$ symbols (and possibly less). In particular, we have that if $|m'| + MaxExt(m) < k_{min}$, then we can avoid any further attempt to extend m' because there is no hope to reach length k_{min} for motifs that have m' as prefix.

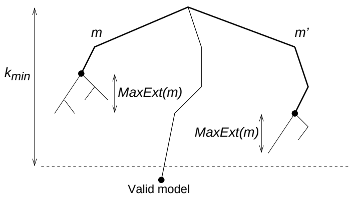

Figure 1: Example where the extension of m' can be avoided, using $MaxExt(m)$, where m is a suffix of m', because $|m'| + MaxExt(m) < k_{min}$.

As we have seen in Algorithm 1, motifs are considered in lexicographical order by a depth-first (virtual) visit of the motif tree $\mathcal{M}$. Every time we stop extending a motif, that is, when we (virtually) backtrack in $\mathcal{M}$, it is either because we found a valid motif of the maximal length, or because the quorum is no longer satisfied ($m\alpha$ does not satisfy the condition at step 2, and we start to consider the next one in lexicographical order). More formally, the analysis the motif $m = \sigma_1, \ldots, \sigma_{|m|}$ with $\sigma_i \in \Sigma$, $\forall i = 1 \ldots |m|$, is abandoned either when m is valid and $|m| = k_{max}$, or m does not satisfy the quorum.

In the first case, m is valid, as are all its prefixes, and $|m| = k_{max}$. No information on the maximal extension of m nor of its prefixes can be of any use because all motifs having a prefix of m as suffix can in general still be extended as much as necessary to reach at least the length k_{min}. For this reason, we set $MaxExt(m) = +\infty$, meaning that m can be extended possibly more than we are computing.

In the second case, m does not satisfy the quorum while all its prefixes do. For reasons that will be clearer later, we chose to only use the maximal extensibility information of motifs of length up to $k_{min} - 1$, hence this case can be subdivided into two subcases. When a motif m cannot be extended anymore and it has not reached the length $k_{min} - 1$, we set $MaxExt(m) = 0$. If the motif has reached a length h between $k_{min} - 1$ and k_{max}, we set $MaxExt(\langle m\alpha|_{k_{min}-1}) = h - (k_{min} - 1)$, where $\langle m\alpha|_{k_{min}-1}$ is the prefix of length $k_{min} - 1$ of $m\alpha$. Since it can be that $MaxExt(\langle m\alpha|_{k_{min}-1})$ had already received some value because a previous extension of $\langle m\alpha|_{k_{min}-1}$ was interrupted, then we change the value of $MaxExt(\langle m\alpha|_{k_{min}-1})$ only if we are increasing it, as maximal extensibility of a motif refers to its longest extension. We assume that all maximal extensibility values are initially set to -1, hence the first attribution to $MaxExt(\langle m\alpha|_{k_{min}-1})$ will always increase its value.

In all the cases above, the algorithm does not consider any further extension of m, and backtracks. This backtracking consists in either replacing the last letter $\sigma_{|m|}$ of m (line 1), or considering a shorter motif which in general shares a prefix with m, if $\sigma_{|m|}$ was the last letter of the alphabet Σ. In this latter case, the whole subtree rooted at the node spelling $\sigma_1 \dots \sigma_{|m|-1}$ has been (virtually) completely visited. Thus, we have all the information necessary to set the value of $MaxExt(\sigma_1 \dots \sigma_{|m|-1})$ according to $MaxExt(x) = 1 + \max_{\alpha\in\Sigma} MaxExt(x\alpha)$, for all valid motifs x such that $|x| < k_{min} - 1$. If the letter $\sigma_{|m|-1}$ was the last of the alphabet, then the backtracking goes further. In that case, also the $MaxExt$ information concerning the word $\sigma_1 \dots \sigma_{|m|-2}$ can be filled in in the same way, and so on as long as we (virtually) climb up in the tree.

As mentioned before, maximal extensibility information can be used for motifs whose extension is being considered and for which this information could actually prevent some useless attempts. Namely, assume we are trying to extend the motif $m = \sigma_1, \sigma_2 \dots, \sigma_{|m|}$. Since the motifs are considered by means of a depth-first search on the virtual motif tree, we obviously do not know the value of $MaxExt(m)$ yet. Moreover, we know $MaxExt(\sigma_2, \dots, \sigma_{|m|})$ only if it lexicographically precedes m, that is, it has already been virtually visited in the motif tree. If this is not the case, we check whether $MaxExt(\sigma_3, \dots, \sigma_{|m|})$ is already known, and so on, possibly until the singleton $\sigma_{|m|}$. If they are all lexicographically greater than m, then no maximal extension information can be used for m, but if for any of them $MaxExt$ is known and it holds that the maximal possible extension is not enough to reach k_{min}, then the information is useful as it guarantees that attempting to further extend m is useless.

Lemma 1 Let $w \in \Sigma^*$. We have $MaxExt(w) \leq MaxExt(v)$ for each v which is a suffix of w.

Proof: Let $MaxExt(w) = k$, there exists $s \in \Sigma^k$ such that the motif ws is valid, that is, it appears in at least q sequences, and no longer string in Σ^* has the same property. Let us now assume that there is a suffix v of w such that $MaxExt(v) = j < k$. Then there exists $t \in \Sigma^j$ with $j < k$, and no longer t, such that the motif vt is valid. However, we know that there exists $s \in \Sigma^k$ such that ws appears in at least q sequences. Since vs is a suffix of ws, and since it satisfies the quorum, then the hypothesis is contradicted. $\square$

A consequence of Lemma 1 is that longer suffixes of m can give us more tight bounds on the maximal extensibility information with respect to shorter ones. Therefore, since we start by checking the longest one, as soon as we find a suffix of m that enables us to state that m is not worth further attempts, then we can stop checking the other (shorter) suffixes. That is, if we find a suffix $|m\rangle_j = \sigma_j, \ldots, \sigma_{|m|}$ of m, with $1 < j \leq |m|$, such that $MaxExt(|m\rangle_j)$ is not enough for m to reach k_{min} because $MaxExt(|m\rangle_j) + |m| < k_{min}$, then we can quit attempting m and all its extensions, and we can consequently update $MaxExt(m)$. On the other hand, if no suffix $|m\rangle_j$ of m is such that $MaxExt(|m\rangle_j) + |m| < k_{min}$, then the maximal extension does not disallow to reach k_{min}. In this case, we have to go on trying to extend m even if it might be the case that it will never reach the minimal length.

The algorithm for single motif extraction using the maximal extensibility information is presented in Algorithm 2. For simplicity, we denote in the same way a node x and the word spelled by the path from the root to x. Moreover, recall that we use $\langle m\alpha|_{k_{min}-1}$ to denote the prefix of $m\alpha$ of length $k_{min} - 1$, and $|x\rangle_{|x|-1}$ to denote the suffix of x of length $|x| - 1$. Finally, with regard to step 3, recall that we assumed that all maximal extensibility values are initially set to -1.

Algorithm 2 Single motif extraction with maximal extensibility information

ExtractSingleMotif(motif m)

1. **for all** $\alpha \in \Sigma$ **do**
2. $x := m\alpha$
3.   **repeat** $x := |x\rangle_{|x|-1}$ **until** ($x = root$ **or** $MaxExt(x) \neq -1$)
4.   **if** $x \neq root$ **and** $MaxExt(x) + |m\alpha| < k_{min}$ **then**
5. $MaxExt(m\alpha) := MaxExt(x)$
6.   stop spelling $m\alpha$ and **continue**
7.   **if** $m\alpha$ is valid **then**
8.   **if** $|m\alpha| \geq k_{min}$ **then** spell out the valid motif
9.   **if** $|m\alpha| < k_{max}$ **then** **ExtractSingleMotif**($m\alpha$)
10.   **else** $MaxExt(\langle m\alpha|_{k_{min}-1}) := +\infty$
11.   **else**
12.   **if** $|m\alpha| < k_{min}$ **then** $MaxExt(m\alpha) := 0$
13.   **else if** $MaxExt(\langle m\alpha|_{k_{min}-1}) < |m\alpha| - (k_{min} - 1)$ **then** $MaxExt(\langle m\alpha|_{k_{min}-1}) := |m\alpha| - (k_{min} - 1)$
14. **if** $|m| < (k_{min} - 1)$ **then** $MaxExt(m) := 1 + \max_{\alpha \in \Sigma} MaxExt(m\alpha)$

2.1.1 Complexity analysis

The time complexity of Algorithm 2 remains the same as for Algorithm 1 in the worst case. Nevertheless, the proposed improvement has (very positive) effects on the average case. Next we show how to compute, in average, the ratio between the number of attempted extensions by RISO and RISOTTO for single motif extraction and compute the limit from which RISOTTO performs better than RISO.

Assume that the dataset has r planted random motifs of size t, where each motif can be extracted with at most e mismatches, and that the remaining text is uniformly random. This assumption captures the fact that we want to analyze the ratio between the number of attempted extensions by RISO and RISOTTO in the context of a dataset with highly correlated sequences (meeting the application requirements to biological datasets).

Let M_i be the random variable that gives the number of extracted motifs of size i with at most e mismatches for the assumed dataset, where $0 \leq i \leq t$. Clearly, we have that $P(M_0 = 1) = 1$ and $P(M_t \geq r) = 1$. The number of attempted extensions by RISO at level

$i > 0$ (when the recursion step is at level i) is given by the random variable

$$E_i = M_{i-1}|\Sigma|,$$

and the total number of attempted extensions for the extraction of a single motif of size k is given by $R_k = \sum_{i=1}^{k} E_i$. On the other hand, RISOTTO will only extend words at level i if they fulfill the maximum extensibility requirement. Therefore the number of attempted extensions by RISOTTO at level i is given by

$$E'_i = M_{i-1}|\Sigma|(1 - p(i)),$$

where $p(i)$ is the probability of a i-word having maximal extensibility information to avoid its extension. Furthermore, the total number of attempted extensions by RISOTTO for the extraction of a single motif of size k is given by $R'_k = \sum_{i=1}^{k} E'_i$.

We conclude that to compute the average value of $\frac{R'_k}{R_k}$ we need to determine the average of the random variables M_i and the values $p(i)$, for $i = 1, \ldots, k$. We proceed by computing the average values of M_i. Clearly, a planted motif of size t has $t - i + 1$ segments of size i (considering overlapping). Observe that the average number of mismatches of the e-occurrences of an extracted motif of size t is given by

$$\overline{e} = \sum_{j=0}^{e} j \frac{\binom{t}{j} (|\Sigma| - 1)^j}{\nu(e, t)}.$$

Hence, if we assume the mismatches to distribute uniformly over the segments, the average number of mismatches of the segments of size i of the e-occurrences is $\overline{e}_i = \frac{i}{t}\overline{e}$. Thus, the motifs extracted at level i due to the planted motifs are all the neighbors differing at most $(e - \overline{e}_i)$ letters from the segments of size i of the planted motifs. Since there are $r(t - i + 1)$ segments of size i, the average number of extracted motifs of size i with at most e mismatches due to the planted motifs is

$$\overline{T}_i = |\Sigma|^i \left(\sum_{j=0}^{r(t-i+1)-1} \left(1 - \frac{\nu(e - \overline{e}_i, i)}{|\Sigma|^i}\right)^j \frac{\nu(e - \overline{e}_i, i)}{|\Sigma|^i} \right).$$

Finally, to determine the average value of M_i, we need to take into account the motifs extracted from the random part of the text, and so, we have

$$\overline{M}_i = \overline{T}_i + (|\Sigma|^i - \overline{T}_i)(1 - \pi_i)$$

where π_i is the probability of a random word of size i not being extracted with quorum q from a set of N sequences. Given that the probability of an e-neighbor of a word of size i not appearing in a random text of size n is

$$\delta(i, e, n) = (1 - 1/|\Sigma|^i)^{(n-i+1)\nu(e,i)} \approx (1 - 1/|\Sigma|^i)^{ni^e|\Sigma|^e},$$

the value of π_i can be computed by the following binomial

$$\pi_i = \sum_{j=0}^{q-1} \binom{N}{j} \delta(i, e, n)^{N-j}(1 - \delta(i, e, n))^j.$$

We finalize by computing the probability $p(i)$. Since the probability of a suffix of a random word being lexicographically smaller than the random word is $\frac{1}{2}$, we have that

$$p(i) = \sum_{j=1}^{i} \frac{1}{2^j} \gamma_{k-i}$$

where γ_{k-i} is the probability of the suffix of size $k-i$ to have information to avoid the extension. Notice that γ_{k-i} is the probability of the suffix of size $k-i$ not being extended to a size greater than $k-1$, and is given by

$$\begin{aligned}\gamma_{k-i} &= \pi_{k-i} + (1-\pi_{k-i})\pi_{k-i+1}^{|\Sigma|} + (1-\pi_{k-i})(1-\pi_{k-i+1}^{|\Sigma|})\pi_{k-i+2}^{|\Sigma|^2} + ... \\ &= \sum_{j=0}^{i-1} \pi_{k-i+j}^{|\Sigma|^j} \prod_{\ell=1}^{j} (1-\pi_{k-i+j-\ell}^{|\Sigma|^{j-\ell}}) .\end{aligned}$$

To understand when RISOTTO starts to provides a gain over RISO, it is important to look to E'_i and E_i. Note that if M_{i-1} is larger than M_i, E'_i will be much smaller than E_i if $p(i)$ is close to 1. Moreover, as soon as random motifs start to disappear, M_{i-1} will be larger than M_i, which happens when π_i is close to 1. Both π_i and $p(i)$ depend tightly of $\delta(i, e, n)$, that is, if $\delta(i, e, n)$ is close to 0, so are π_i and $p(i)$, and if $\delta(i, e, n)$ is close to 1, so are π_i and $p(i)$. Since $\delta(i, e, n)$ behaves like a Dirac cumulative function for large values of n, that is, it jumps very fast from 0 to 1, we just need to solve the equation $\delta(i, e, n) = 1/2$ for the variable i to grasp when RISOTTO starts to be faster than RISO, which is just slightly before the solution. The solution of that equation is the fixed point of the following function

$$f(x) = (-\log(1 - \frac{1}{2^{|\Sigma|^e x^e n}})/\log(\Sigma)).$$

Given that $f(x)$ is contractive, that is, its derivative function takes values in the interval $(-1, 1)$, the fixed point can be computed by iterating f over an initial value. Finally, notice that the fixed point increases with the values of e, n and Σ.

With the previous analysis, we have all the machinery necessary for computing the ratio between the expected number of attempted extensions between RISO and RISOTTO, as well as, from which point RISOTTO performs better than RISO. As an example, the ratio between the expected number of extensions attempted by RISOTTO and RISO for a dataset consisting of $N = 100$ sequences of size $n = 1000$ where we planted $r = 1$ motif of size $t = k = 5..20$, with up to $e = 2$ mismatches, and quorum $q = 100$, is given in Figure 2. For the dataset considered, the fixed point for $f(x)$ is $x = 10.6616$.

3 Structured motif extraction

A *structured motif* is a pair (m, d) where $m = (m_i)_{1 \leq i \leq p}$ is a p-tuple of single motifs and $d = (d_{min_i}, d_{max_i})_{1 \leq i < p}$ is a $(p-1)$-tuple of pairs, denoting $p-1$ intervals of distance between the p single motifs. Each element m_i of a structured motif is called a *box* and its minimal and maximal length denoted by k_{min_i} and k_{max_i}, respectively. The *structured motif extraction problem* takes as parameters N input sequences, a quorum $q \leq N$, p maximal error rates $(e_i)_{i \leq 1 \leq p}$ (one for each of the p boxes), p minimal and maximal lengths $(k_{min_i})_{i \leq 1 \leq p}$ and $(k_{max_i})_{i \leq 1 \leq p}$ (one for each of the p boxes), and $p-1$ intervals of distance $(d_{min_i}, d_{max_i})_{i \leq 1 \leq p-1}$

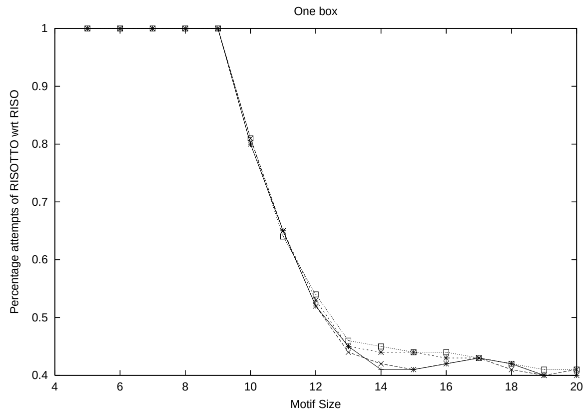

Figure 2: Ratio between the expected number of extensions attempted by RISOTTO and RISO (cf Figure 3 at Section 4 to compare theoretical with experimental results obtained in the same set).

(one for each pair of consecutive boxes). Given these parameters, the problem consists in searching for the contents of the boxes, that is the motifs, that have the structure defined by the parameters above and that satisfy the quorum. The algorithm for single motif extraction introduced in [13] is the ancestor of a couple of others [3, 10] that infer structured motifs. The optimisation introduced in this paper can be applied to any of them.

In a few words, the algorithm first builds the factor tree $\mathcal{T}$ of the input sequences, then it searches for all valid motifs of length at least k_{min} and up to k_{max} (as in [13]) and, after updating the data structure (see [3] for details), checks whether there is a second valid motif (again as in [13]) with the required interval between them. More formally, the algorithm is described by Algorithm 3 assuming for simplicity that $p = 2$, where the motif m is the one whose extension is being attempted, and the value i indicates whether we are dealing with the first or the second box. Finally, λ denotes the empty word.

Algorithm 3 Structured motif extraction

ExtractStructuredMotif(motif m, box i)
1. **for all** $\alpha \in \Sigma$ **do**
2. **if** $m\alpha$ is valid **then**
3. **if** $|m\alpha| \geq k_{min_i}$ **then**
4. **if** $i = 2$ **then** spell out the valid motif
5. **else** update $\mathcal{T}$ to **ExtractStructuredMotif**$(\lambda, 2)$
6. **if** $|m\alpha| < k_{max_i}$ **then** **ExtractStructuredMotif**$(m\alpha, i)$

3.1 Using maximal extensibility of factors

In the case of structured motifs, the maximal extensibility information for the first box of a motif should be updated as described in Section 2.1. However, any failure in attempting to extend a motif during the search of a second box cannot update any value of $MaxExt$ because it refers only to parts of the text that follow a specific first box at a specific distance. In fact, when a first box m_1 of a structured motif is fixed at any given step, the maximal extensibility information that concerns the whole sequence is in general an upper bound on the maximal extensibility of fragments of the sequence that are at a given distance from the occurrences of m_1. Given this observation, a possibility is to use the maximal extensibility information of the first box when searching and trying to extend a second box. Another possibility, while attempting to find a motif for the second box, is to compute and store tighter maximal extensibility information which we can use for the second box being attempted as long as the first box is fixed. In the following, we only address the first alternative, that is, only the first box stores extensibility information. The conditions needed for our optimisation to be applicable in the case of structured motifs may hold even more frequently than in the case of single motifs. In fact, since the search for a valid motif as second box is made after a valid motif for the first box is found, maximal extensibility information may be known also for the whole motif whose extension is attempted and not just for its prefixes. In other words, it may happen that when Algorithm 3 is called with parameters m and 2, the value of $MaxExt(m)$ is already known. Proper suffixes are thus not the only candidates to give useful information when we are trying to find a motif for the second box. The extensibility information can be used as for the case of single motifs except that one has to deal with different error rates among boxes. Indeed, e_2 must be less than or equal to e_1 in order for the extensibility information to be useful for the second box. Otherwise, the maximal extensibility information stored for the first box may be too restrictive, and if it is used, it may cancel the extension of valid motifs. The algorithm for structured motif extraction using the maximal extensibility information is presented in Algorithm 4. Similarly to the case of single motif extraction, the time complexity

Algorithm 4 Structured motif extraction with maximal extensibility information

ExtractStructuredMotif(motif m, box i)
for all $\alpha \in \Sigma$ **do**
  if $i = 1$ **or** $e_2 \leq e_1$ **then**
    $x := m\alpha$
    while $(x \neq root$ **or** $MaxExt(x) = -1)$ $x := |x\rangle_{|x|-1}$
    if $x \neq root$ **and** $MaxExt(x) + |m\alpha| < k_{min_i}$ **then**
      if $i = 1$ **then** $MaxExt(m\alpha) := MaxExt(x)$
      stop spelling $m\alpha$ and **continue**
  if $m\alpha$ is valid **then**
    if $|m\alpha| \geq k_{min_i}$ **then**
      if $i = 2$ **then** spell out the valid motif
      else follow box-links and update $\mathcal{T}$ to **ExtractStructuredMotif**$(\lambda, 2)$
    if $|m\alpha| < k_{max_i}$ **then ExtractStructuredMotif**$(m\alpha, i)$
    else if $i = 1$ **then** $MaxExt(\langle m\alpha|_{k_{min_1}-1}) := +\infty$
  else if $i = 1$ **then**
    if $|m\alpha| < k_{min_1}$ **then** $MaxExt(m\alpha) := 0$
    else if $MaxExt(\langle m\alpha|_{k_{min_1}-1}) < |m\alpha| - (k_{min_1} - 1)$ **then** $MaxExt(\langle m\alpha|_{k_{min_1}-1}) := |m\alpha| - (k_{min_1} - 1)$
if $i = 1$ **and** $|m| < (k_{min_1} - 1)$ **then** $MaxExt(m) := 1 + \max_{\alpha \in \Sigma} MaxExt(m\alpha)$

of Algorithm 4 remains the same as for Algorithm 3 in the worst case, and the improvement proposed accounts only for the average case, as we shall verify in the next section.

4 Implementation and experimental results

In order to verify the improvement proposed in this paper, a C implementation of the maximal extensibility algorithm, called RISOTTO[1], was made. The new implementation was tested against a C implementation of the algorithm presented in [3] and called RISO. The results of the experiments we made show a sensible improvement for both single and structured motif extraction when using maximal extensibility information. As we shall see in this section, maximal extensibility may cost some extra space, which is a delicate issue for large datasets, but it can definitely save some hopeless visits, and in general it results very efficient.

We start with some considerations concerning the storage of extensibility information. As we have seen in Section 2.1, due to the order in which motifs are considered, we have that only certain subwords of motifs can give useful information concerning maximal extensibility, namely, those that are lexicographically smaller. Since no motif is smaller than itself, we actually only use the $MaxExt$ information of motifs that are shorter than the current one, that is, they are proper suffixes. Therefore, since the condition to check is whether or not we can hope to reach the k_{min} length, then we make use of the $MaxExt$ data only for strings of length at most $k_{min} - 1$. Hence, it is not necessary to store this information for motifs that have length k_{min} or more for the purpose mentioned above. Let us now discuss how much space is required to store the extensibility information until level $k_{min} - 1$. We say that a tree is *uncompact complete* if it is a trie where all possible nodes are present. There is thus no arc whose label contains more than one letter. A previous result [1] makes use of some statistical analysis for stating that a suffix tree of a text of length n is expected to be uncompact complete at the $log_{|\Sigma|}(n)$ top levels, where Σ is the alphabet of the text. This fact suggests a model to store extensibility information: a static data structure to keep the $MaxExt$ values until level $log_{|\Sigma|}(n)$, and a dynamic structure for deeper levels. Since we are interested in the DNA alphabet (composed of the four nucleotides A, C, G, and T), then we have that our suffix tree is uncompact complete at the top $log_4(n)$ levels where n is the size of the input sequence s. The function $log_4(n)$ reaches 10 for $n \approx 10^6$, it is greater than 11 for $n = 10^7$, it is more than 13 for $n = 10^8$, and nearly 15 for $n = 10^9$. These values correspond to reasonable values for the minimal length k_{min} of the motif, and they are reached for values n of the text size corresponding to quite big datasets. In the RISOTTO implementation, we took all the observations above into consideration. Since k_{min} has to be relatively small for our approach to be tractable spacewise, we considered only 1 byte (a char in C) to store $MaxExt$ values. In this case, extensibility values must be less than 256, which is quite reasonable. To build a static data structure to store such values until level z, we need $z + 1$ 1-byte arrays, where the j-th array has size $|\Sigma|^j$ with $0 \leq j \leq z$. Therefore, for the case of DNA, the total amount of memory required is $\frac{4^{z+1}-1}{3}$ bytes. This function gives us values of 1.3MB for $z = 10$, 5.3MB for $z = 11$, 85.3MB for $z = 13$, and 1.3GB for $z = 15$. In our experiments, we achieved an optimum trade-off between memory allocation/management and maximal extensibility gain when $z = 10$. Taking this observation into account, we only allocate values for $MaxExt$ until level $z = \min\{10, k_{min} - 1\}$, even for large values of k_{min}, and disregard deeper levels as well as the dynamic data structure mentioned above. Nevertheless, we allowed this z level to be an implementation parameter. In the end, considering $z = \min\{10, k_{min} - 1\}$, RISOTTO requires at most 1.3MB more that RISO for DNA databases, being more than twice faster as we shall see next.

[1] RISOTTO is available at `http://algos.inesc-id.pt/~asmc/software/riso.html`.

To test maximal extensibility performance we used several randomly generated (with a uniform distribution over the four letters size DNA alphabet) synthetic datasets with planted structured motifs. Each dataset consists of 100 sequences of size 1000 where we planted one motif, possibly structured into several boxes, with 2 mismatches per box. We ran both RISO and RISOTTO requiring a quorum $q = 100$ and at most 2 mismatches per box so that the output contains at least the planted motif. For each dataset, we made several runs for increasing lengths of the motifs. In particular, given the number of boxes of the structured motifs (in our tests there are p boxes for $p = 1, \ldots, 4$), we have increased the size of the boxes ranging from 5 to 20. As a result, the total motifs size (without counting the gaps) ranges from 5 to 80. For each p (number of boxes), we have plotted in Figure 3, against the size of the motif (x axis), the ratio between the number of extensions attempted by RISOTTO and those by RISO (y axis). Given than RISOTTO only saves useless attempts, this equals the

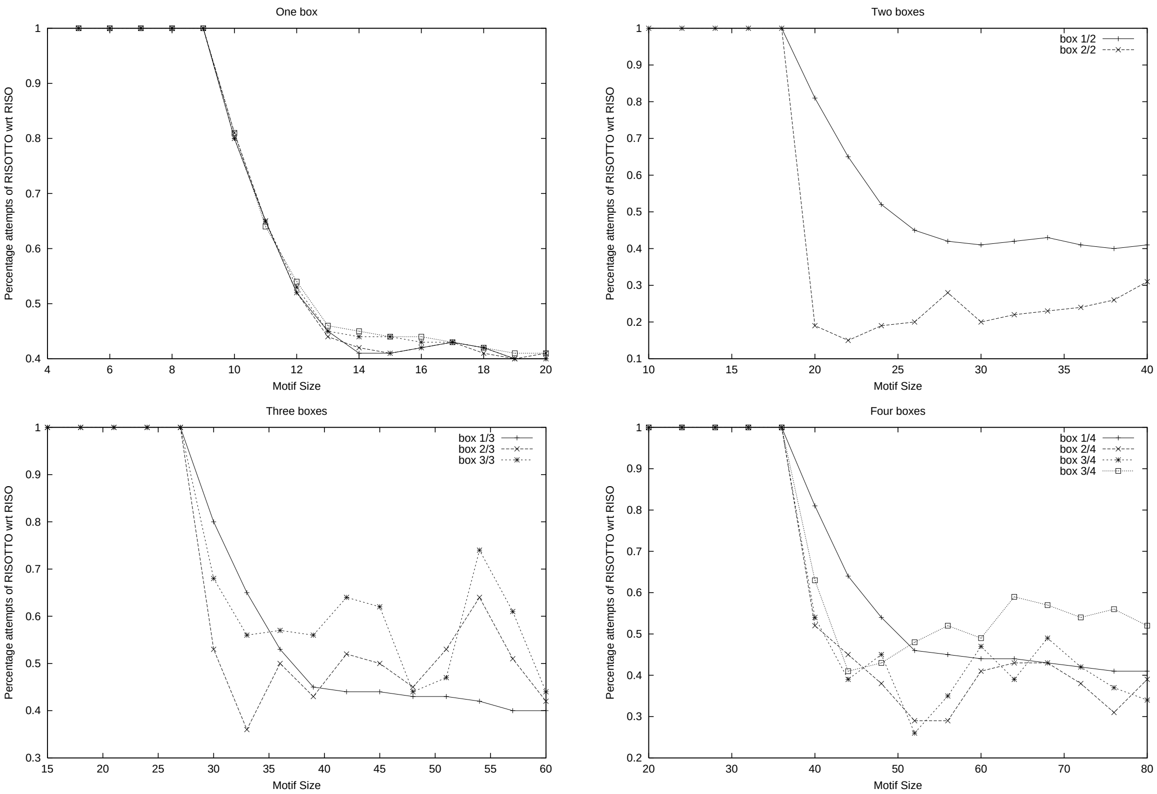

Figure 3: Ratio between the number of extensions attempted by RISOTTO and RISO (cf Figure 2 at Section 2.1.1 to compare theoretical with experimental results obtained in the same set).

percentage of saved calls of the recursive procedure. For one box (Figure 3 top left) we have depicted the results for several runs, while for two, three and four boxes (Figure 3 top right and bottom) there are one curve for the inference of each box of the structured model. As one would expect, the attempts saved are more when the length of the motif increases and, in particular, the improvement starts when the length of the box is about 10 (this value depends in general from the input sequence and the alphabet size). For one box (see Figure 3 top left), the number of attempted extension of RISOTTO decreases fast to 40% with respect to

RISO (for growing values of the length of the motifs). Even better results, getting as good as attempting only 20% of the extensions of RISO, were achieved when extracting an i-th box with $2 \leq i \leq p$ (see Figure 3 (top right and bottom)). Moreover, we present the ratio of speed performance of the computation of RISOTTO with respect to that of RISO. This is shown for all tests together in Figure 4 for all possible sizes of the boxes. One can see that

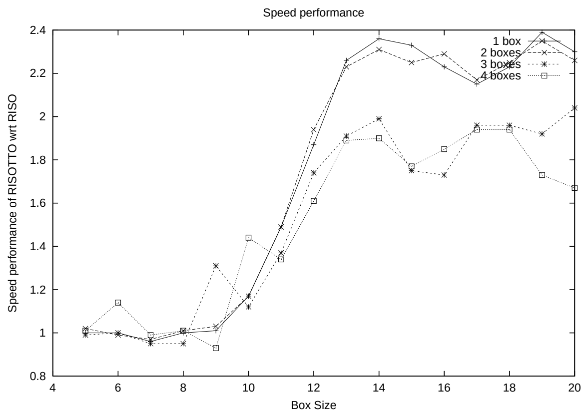

Figure 4: Ratio between performance of RISOTTO and RISO.

the best relative performance is achieved for the first boxes (that is where it is more needed because the search space is very large and noisy), where RISOTTO is up to 2.4 faster than RISO.

Finally, in [11] a challenging problem was launched that concerned finding all single motifs of length 15 with at most 4 mismatches in 20 texts of size 600. We ran both RISO and RISOTTO on such instances. We observe a speedup of 1.6 of RISOTTO over RISO. We actually believe that a true challenge should involve texts of larger size. Therefore, we ran tests with the same parameters (length 15 and at most 4 mismatches) on larger input sequences. The results confirm the 1.6 speedup for sequences of length 700 and 800, 1.3 speedup for length 900, and then the speedup decreases, but the time required by RISOTTO is always lower than for RISO.

5 Conclusions and further work

We presented a new algorithm for the extraction of structured motifs in DNA sequences, improving what is, to our knowledge, the best known exact algorithm for extracting structured motifs. The improvement consists in storing information concerning maximal extensibility of factors in order to avoid trying to extend hopeless motifs. Experimental results show that the improvement works for large motifs, achieving 40% of the computational time. In terms of space, a trade off between memory allocation/management and maximal extensibility gain was made, leading up to 1.3MB of memory cost for storing the extensibility information.

The amount of saved visits achieved by extensibility improvement, and their impact on the performance of motif extraction, depends on the instances and the values of the parameters. Some further studies on real data or some possible preprocessing could be helpful in detecting which are the ranges of values of the parameters for which this modification is more suitable. We also plan to apply RISOTTO to the benchmarks of [6], given that RISOTTO performs better than some of the best tools of such assessment (actually inspired by the SMILE tool that inspired RISO).

Finally, as we have seen, the structured motif extraction is made considering motifs according to a depth-first visit to the motifs tree. It is intuitive to see that in the case of a breadth-first search visit [14], the maximal extensibility information could be available for all shorter submotifs of the one being extended. This could result in a possible increase of the number of saved visits. It could therefore be interesting to perform the modification presented in this paper for motif extraction algorithms that consider strings in an order corresponding to a breadth-first search visit of the tree of the motifs.

References

[1] J. Allali. Structures d'indexation: les arbres des facteurs. Memoire de maitrise, University of Marne-la-Vallée, 2000.

[2] J. Buhler and M. Tompa. Finding motifs using random projections. *J. Comp. Bio.*, 9(2):225–242, 2002.

[3] A. M. Carvalho, A. T. Freitas, A. L. Oliveira, and M.-F. Sagot. A highly scalable algorithm for the extraction of cis-regulatory regions. In *Proc. APBC'05*, pages 273–282. Imperial College Press, 2005.

[4] M. Crochemore and T. Lecroq. Pattern matching and text compression algorithms. *ACM Computing Surveys*, 28(1):39–41, 1996.

[5] E. Eskin and P. A. Pevzner. Finding composite regulatory patterns in DNA sequences. *Bioinformatics*, 18(1):354–363, 2002.

[6] M. Tompa et al. Assessing computational tools for the discovery of transcription factor binding sites. *Nature Biotechnology*, 23(1):137–144, 2005.

[7] D. Guha-Thakurta and G. D. Stormo. Identifying target sites for cooperatively binding factors. *Bioinformatics*, 17:608–621, 2001.

[8] X. Liu, D. L. Brutlag, and J. S. Liu. Bioprospector : discovering conserved DNA motifs in upstream regulator regions of co-expressed genes. In *Proc. PSB'01*, pages 127–138, 2001.

[9] M. Lothaire. *Applied Combinatorics on words*. Cambridge University Press, 2005.

[10] L. Marsan and M.-F. Sagot. Algorithms for extracting structured motifs using a suffix tree with application to promoter and regulatory consensus identification. *J. Comp. Bio.*, 7:345–360, 2001.

[11] P. A. Pevzner and S. H. Sze. Combinatorial algorithm for finding subtle signals in dna sequences. In *Proc. ISMB'00*, pages 269–278, 2000.

[12] A. Mukherjee R. V. Satya. New algorithms for finding monad patterns in dna sequences. In Alberto Apostolico and Massimo Melucci, editors, *Proc. SPIRE'04*, volume 3246 of *LNCS*, pages 273–285. Spriger-Verlag, 2004.

[13] M.-F. Sagot. Spelling Approximate repeated or common motifs using a suffix tree. In C.L. Lucchesi and A.V. Moura, editors, *Proc. Latin'98*, volume 1380, pages 111–127, 1998. LNCS.

[14] M.-F. Sagot, A. Viari, and H. Soldano. Multiple sequence comparison: a peptide matching approach. In *Proc. CPM'95*, volume 937, pages 366–385, 1995. LNCS.

[15] E. Segal, Y. Barash, I. Simon, N. Friedman, and D. Koller. A discriminative model for identifying spatial cis-regulatory modules. In *Proc. RECOMB'04*, pages 141–149, 2004.

[16] R. Sharan, I. Ovcharenko, A. Ben-Hur, and R. M. Karp. Creme: a framework for identifying cis-regulatory modules in human-mouse conserved segments. *Bioinformatics*, 19(Suppl 1):i283–i291, 2003.