Application of suffix trees for the acquisition of common motifs with gaps in a set of strings

Pavlos Antoniou, Maxime Crochemore, Costas Iliopoulos, Pierre Peterlongo

Application of suffix trees for the acquisition of common motifs with gaps in a set of strings

Pavlos Antoniou[1], Maxime Crochemore[2], Costas S. Iliopoulos[1], Pierre Peterlongo[2]

[1] Dept. of Computer Science, King's College London, London WC2R 2LS, England, UK
[2] Institut Gaspard-Monge,
Laboratoire d'informatique Université de Marne-la-Vallée, 77454 Marne-la-Vallée CEDEX 2, France

Abstract. The inference of common motifs in a set of strings is a well-known problem with many applications in biological sciences. We study a new variant of this problem that offers a solution with the added flexibility in the length of the common motifs to be found. We present algorithms that allow stretching of the length of the motifs as well as elasticity in the length of gaps between the motifs. The main data structure used in these algorithms is the suffix tree.

1 Introduction

The problem of finding common motifs with gaps is that of finding common patterns that occur in every string of the set $S = \{S_1, S_2, \ldots, S_r\}$, in various locations. In each string, these patterns are separated by don't care symbols concentrated in distinct parts of contiguous positions, which form the gaps between the motifs.

Finding motifs with gaps is a problem with many applications in biology. In gene expression, transcription factors bind to DNA sequences at specific locations, (the binding sites), which are specific short sequences of nucleotides. These binding sites share common patterns,which are the common motifs [12].

Multiple Sequence Alignment (MSA) [14] was one of the first methods employed for finding common motifs in DNA sequences. The problem of MSA is NP-complete and as such this method is computationally demanding and therefore in practice, it can only be applied to a small number of sequences.

Carvalho *et al.* in [2] have developed an implementation called RISOTTO which extracts motifs in DNA, RNA, protein molecules and from whole genomes. The algorithm extracts single and structured motifs. Structured motifs are motifs composed of several disjoint single motifs and are useful in the detection of binding sites and cis-regulatory modules in genomic sequences [9]. In higher eukaryotes, the binding sites for the transcription factors are organized in short sequence units called cis-regulatory modules. In [9], the authors created an exponential time algorithm, that uses a novel data structure ,called box-link, to store information about the conserved regions in the sequences, in order to detect these binding sites.

The term "motif" has been given a new definition in [3], in the context of metabolic networks. In this context, motifs are defined as small, repeated and evolutionary conserved subnetworks. They do not function in isolation, and can be overlapping and nested. The authors have used graph models to detect this type of motifs.

The suffix tree has been the main data structure used in string algorithms for finding common motifs in sequences [4, 5, 8, 11]. Marsan and Sagot in [13] developed two exact algorithms using suffix trees to extract conserved structured motifs from a set of DNA sequences. Their algorithms, take into consideration the structure of the binding sites and use the suffix tree to extract site consensus, such as promoter sequences from non-coding sequences. Sagot describes two algorithms in [15] using suffix trees; the first one uses a suffix

tree to extract repeated motifs from one sequence and the second one uses a generalized suffix tree to extract the common motifs of a set of sequences allowing both mismatches and gaps. Crochemore *et al.* [6] describe the notion of the basis of the motifs, from which all the other ones are generated, in an attempt to avoid the combinatorial explosion in the number of motifs. In [11], an algorithm was proposed for finding common motifs with gaps in a set of strings but the motifs were constrained to patterns that had the same length.

We present two algorithms to find the common motifs with gaps in a set of strings, with the patterns that comprise the motifs having variable length. Both algorithms rely on the data structures of the suffix trie and the reverse common trie. The reason behind using these data structures is the fact that the common suffix trie can help us find the right part of repeats in strings and the reverse suffix trie can give us the left part of repeats. Considering a motif consisting of two parts separated by a block of don't cares (the gap), by combining the two tries we can acquire the right and left parts of the motif. Both algorithms require $O(nrm + km)$ time, where where n is the total length of the sequences, r is the number of the sequences, m is the number of nodes in the reverse common trie and k is the number of times we combine a right part with a left part of the string to form a motif.

In section 2, we formally introduce two variants of the common motif problem. In section 3, we describe the two algorithms beginning with the preprocessing steps in section 3.1. In subsection 3.2, we present an algorithm for the problem of finding common motifs with fixed gaps. In subsection 3.3, we describe the algorithm for the problem of finding common motifs with variable gaps. Finally in section 4, an analysis of the complexity of the two proposed solutions is presented.

2 Basic definitions

A string s of length n is a concatenation of n symbols $s_1 s_2 \cdots s_n$, where $s_i \in \Sigma$ for $1 \leq i \leq n$; ϵ is the empty string, a sequence of $\emptyset$ symbols. A string w is a *substring* of s, if $s = uwv$ for $u, v \in \Sigma^*$; in this case we also say that the string w occurs at position $|u| + 1$ of the string s. We denote by $s[i..j]$ with $1 \leq i \leq j \leq n$, the word $s_i s_{i+1} \cdots s_j$ of s. A string w is a *prefix* of s, if $s = wu$ for $u \in \Sigma^*$. Similarly, w is a *suffix* of s, if $s = uw$ for $u \in \Sigma^*$. The *don't care* symbol is a special symbol that matches any other symbol of Σ and it is denoted by $*$.

We consider two variants of the problem of computing common motifs with gaps that differ in the constraints set on the the gaps. In the first variant of the problem, we compute motifs with fixed gaps and in the second variant we compute motifs with variable gaps.

Common Motifs with Fixed Gaps (abbreviated comfig): Given a set of strings $\{S_1, S_2, \ldots, S_r\}$ and integers p, q, d, we are looking for non-empty strings ("motifs") $B_1, B_2, \ldots, B_m$ of maximal size, where each B_i cannot be extended by one symbol or more (to the right or to the left), such that:

$$B_1 *^d B_2 *^d \ldots,^d B_m \text{ occurs in } S_i, \text{ for all } i \in \{1..r\} \text{ with } p \leq |B_j| \leq q \text{ for all } j \in \{1..m\}$$

For example, given a set of three strings S:

S_1= A C G T A C G G A T G C C T C A A A
S_2= G A C C T A C C G A G C G C T C G T T A A
S_3= A C T T A C T G A T T T C T C A A

and integer $p = 1$, gap length $d = 1$ and maximum motif size $q = 4$, we get the following motifs that satisfy the required criteria:

S_1= A C G T A C G G A T G C C T C A A A
S_2= G A C C T A C C G A G C G C T C G T T A A
S_3= A C T T A C T G A T T T C T C A A

This gapped motif must occur at least once in each of the input strings. Note that an instance of this problem is the calculation of the motifs when gap $d = 0$.

In the second variant of the problem, we look for motifs with maximal size and minimum gap sizes. The size of the gaps, is not known and it can be variable.

Common Motifs with Minumum Variable Gaps (abbreviated comvag): Given a set of strings $\{S_1, S_2, \ldots, S_r\}$ and integers p, q, the problem consists of finding for non-empty strings ("motifs") $B_1, B_2, \ldots, B_m$ of maximal size, where each B_i cannot be extended by one symbol or more (to the right or to the left), such that:

$$B_1 *^{d_1} B_2 *^{d_2} \ldots^{d_m} B_m \text{ occurs in } S_i, \text{ for all } i \in \{1..r\}, \text{ with } p \leq |B_j| \leq q, \text{ for all } j \in \{1..m\}$$

For example, given the same set of strings S and integer $p = 1$ and $q = 4$, we get the following motifs that satisfy the required criteria:

S_1= A C G T A C G G A T G C C T C A A A
S_2= G A C C T A C C G A G C G C T C G T T A A
S_3= A C T T A C T G A T T T C T C A A

3 Algorithms

We are going to present algorithms for the two variants: comfig and comvag. Both algorithms require identical preprocessing. During preprocessing we will create the basic data structures of the algorithms; namely the common segments trie, the reverse segments trie and two stacks based on the two tries.

3.1 Preprocessing Steps

The problem of finding common motifs with gaps, is similar to finding the maximal repeats present in strings with contiguous don't care symbols between them, which form the gaps. A motif will be considered as a *maximal repeat*, if it can not be extended to the left nor to the right without losing one of its common occurrences.

An algorithm was developed in [7] that makes use of suffix trees to find the longest repeats with a block of don't cares in one string. The algorithm constructs the suffix tree for a string x, and the suffix tree of the reverse string $\overleftarrow{x}$. The first suffix tree is used to construct the right part of the repeat and the second is used to construct the left part.

Similarly to [7], we will construct the trie of common segments of the input set of strings which will give us the right part of the repeats present in all the strings, i.e the right part of the motifs. Then, we will construct the reverse suffix trie, in order to have the left part of the repeats. These repeats, once merged, will constitute the new right part of a longer repeat, and can be further extended with other parts of the strings to create longer repeats.

Constructing the common segments trie: The trie of common segments $T(S)$ is the trie of all the common suffixes of the strings in set S. Each leaf in $T(S)$ represents a suffix w[i... n]. Each edge in $T(S)$ is labeled with a nonempty substring of all the strings in set S. We refer to a substring spelled by the path from the root to a node u as the label of u and we denote it as $l(u)$.

STEP 1
We built the suffix tree for the first sequence S_1. The suffix tree can be built in linear time using a number of algorithms described in [4, 5, 8, 11]. Apostolico presents a comprehensive review for the algorithms and characteristics of suffix trees in [1] and [10].
STEP 2
In each node of the tree we add two arrays of size r. The first array A is used to keep track of the number of different strings that each node (suffix) belongs to; the second array B is used to hold the rightmost position of the pattern ending in that node. We save the rightmost position of the pattern in array B, so that we can be closer to the left part of the repeat which will be given to us by the reverse trie.
STEP 3
We mark the arrays A and B at each node for the rest of the strings in S_i. Whenever we visit a node that belongs to S_i and S_1, we mark the array A of that node at the position which corresponds to the index i of string S_i and we mark the rightmost position of its label in B. If a node of a string S_i is not already in the suffix tree S_1 we ignore it.
STEP 4
We traverse the suffix tree checking to see, if the array A at each node is marked in all of its positions, i.e if that particular node belongs to all the strings in the set. Whenever, we find a node that does not have all its bit array positions marked, we delete it and its children, thus creating a cutoff point at that node. By deleting these nodes, we no longer have a suffix tree, but instead we acquire the common segments trie $T(S)$ of the strings. Figure 1a presents an example of a common segments trie for the three strings $S_1 = ACTAGAT$, $S_2 = GTCTACATC$ and $S_3 = CTATATG$.

Constructing the reverse common trie: We create the reverse common segments trie $(\overleftarrow{T(S)})$ based on $T(S)$. Given a node u in trie $T(S)$ and its label $l(u)$, at each node v of the trie $(\overleftarrow{T(S)})$, we enter the reverse label $\overleftarrow{l(u)}$ as label $l(v)$ and the leftmost position of the reverse label $\overleftarrow{l(u)}$ in the B array of that position. By doing that, for every node of $T(S)$, we end up with the reverse common suffix trie $(\overleftarrow{T(S)})$. Figure 1b presents an example of a reverse common trie for the trie of Figure 1a.

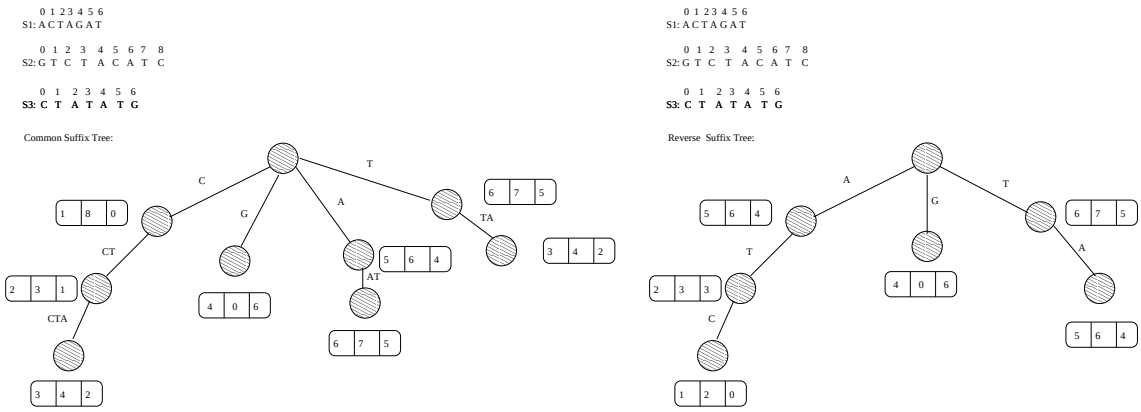
Constructing the stacks: We insert the nodes of the $T(S)$ and $(\overleftarrow{T(S)})$ in two stacks L and M. This way we avoid having to read the tries many times while we do our computations and we can store the nodes in the stacks sorted. In the first stack L, we insert each node of $T(S)$ from leaf to node, (depth first), so that if we read the contents of the stack from top to bottom it would be like we are doing a pre-order traversal of the tree. Inversely, in stack M we add the nodes of $(\overleftarrow{T(S)})$ from root to leaves in such a way, that if we read the contents of the stack M, from top to bottom, it would be like a depth first traversal of the tree.

3.2 Common Motifs with Fixed Gaps

Algorithm 1 solves the COMFIG problem and produces the common motifs with fixed gaps of the strings in the set S. The input of the algorithm are the the constraints p, q, the gap d and the two stacks L and M.

The main idea of the algorithm is to extend the labels of the nodes in stack L with the labels of stack M.

STEP 1
Given a node u of stack L, we either, extend the label $l(u)$ with the label $l(v)$ of v sequentially, and combine nodes u and v (lines 17-19), *or*, we extend label $l(u)$ by first adding the



(a) Common segments trie (b) Reverse common trie

Fig. 1. The pruned common trie and the reverse common trie of the example for fixed gaps

Stack L	Labels
1, 8, 0	C
2, 3, 1	CT
3, 4, 2	CTA
4, 0, 6	G
5, 6, 4	A
6, 7, 5	AT
6, 7, 5	T
3, 4, 2	TA

(a)

Stack L	Labels
3, 10, 2	C
4, 5, 3	CT
5, 6, 4	CTA
6, 2, 8	G
7, 8, 6	A
8, 9, 7	AT
8, 9, 7	T
5, 6, 4	TA

(b)

Stack M	Labels
1, 2, 0	C
2, 3, 3	T
5, 6, 4	A
4, 0, 6	G
5, 6, 4	A
6, 7, 5	T

(c)

Fig. 2. Figure (a) presents the stack L. We add the nodes from the common suffix tree in stack L from the leaves to the nodes in a depth first manner so the top of the stack would be the root of the tree. Figure (b) presents stack L after adding the gap length d in each position. Figure (c) presents the M stack which contains the nodes of the reverse trie.

gap d to $B(u)$ and then the label $l(v)$ and node v(lines 20-22). The former case is preferable as it extends the length of the repeat with no gaps.

Step 2

Whenever we combine two nodes u and v we repeat this procedure using the new combined node as a starting node until we cannot extend any more.

For example, given the three example strings of subsection 3.1 and gap $d = 1$, we construct the common segments trie $T(S)$ (Fig. 1a), the reverse trie $(\overleftarrow{T(S)})$ (Fig. 1b), and stacks L and M (Fig. 2). The algorithm starts with the first element in L which is the node with label $l(u) = C$ and $B(u) = \{1, 8, 0\}$, as shown in Fig. 2.

First, we try to extend this label with a label from stack M with no gaps in between. We add 1 to the values of $B(u)$ and we have $B(u) = \{2, 9, 1\}$. There is no match of this array with any arrays of stack M, so, we continue by adding the gap length $d + 1$ to array $B(u)$.

Fig. 2(b), shows the resulting B arrays, after we added gap $d+1=2$ to each position. So, $B(u)$ now becomes $\{3, 10, 2\}$. Again there is no match in stack M, thus we continue with the next element of stack L until we find a match.

Algorithm 1 Finding motifs with fixed gaps COMFIG

Input: p, q, d, node u, node v, arrays $B_1[r]$,$B_2[r]$, stacks L and M, r, arrays $match$, $match2$.
Output: Motifs u_1*gap of size d*u_2*gap of size d*u_3*$\dots u_m$

```
1: while L! = empty do
2:   u = pop(L);
3:   while M! = empty do
4:     v = (M) → top
5:     for i = 0 to r do
6:       B1[i] = Bu[i] + 1
7:       B2[i] = Bu[i] + d + 1
8:     end for
9:     for i = 0 to r do
10:      if Bv[i] == B1[i] then
11:        match + +;
12:      end if
13:      if Bv[i] == B2[i] then
14:        match2 + +;
15:      end if
16:    end for
17:    if match == r then
18:      motif = l(u) + l(v)
19:      if |motif| ≤ q then
20:        u = motif
21:      end if
22:    else if match2 == r then
23:      motif = l(u)+|d|(∗)+l(v)
24:      if |motif| ≤ q then
25:        u = motif
26:      end if
27:    end if
28:    v = v → next
29:  end while
30: end while
```

We won't find a match until we reach the third element of stack L with $l(u) = CTA$ and $B(u) = \{3, 4, 2\}$. Adding 1 to the positions of $B(u)$ does not give a match, but adding $d+1$ to $B(u)$, $B(u)$ becomes $\{5, 6, 4\}$ which is matched in stack M and has label A.

So, $l(u)$ becomes $CTA(*)A$ and $B(u) = \{5, 6, 4\}$. Adding 1 to the positions of $B(u)$ gives a match from stack M which has $l(v) = T$. We add this label to our current label $l(u)$ and the motif becomes:

S_1=A C T A G A T
S_2= G T C T A C A T C
S_3= C T A T A T G

We can't extend this motif any more because we can't find a match either by adding 1 or $d+1$ to $B(u)$. So, we stop the procedure and report the motif $CTA(*)AT$. Choosing subsequently the rest of the motifs as starting points from L, we will produce all possible motifs present at the input strings.

3.3 Computing common motifs with minimum variable gaps

Algorithm 2 solves the COMVAG problem and will produce the common motifs with minimum variable gaps of the input strings . The input of the algorithm are the constraints p, q and the two stacks L and M.

STEP 1
We read all the nodes u from stack L in succession, and match them with the first node v in stack M, read from top to bottom, that has the smallest values in its position of its array $B(v)$, which are larger than the corresponding positions of $B(u)$.
STEP 2
We combine the two nodes adding the gaps between them, gaps that correspond to the difference of their positions $(B(v) - B(u) - 1)$.
STEP 3
Using this new combined node, which has $B(u) = B(v)$, we search for the next node v from stack M that has larger elements in its B array than the elements of $B(u)$. If we find one, we add this new node v to the previous node, thus adding a third substring to the motif, taking again into consideration the gaps between the suffixes.

Algorithm 2 Finding motifs with variable gaps COMVAG

Require: $T(S)$, $(\overleftarrow{T(S)})$, p, q, node u, node v, stacks L and M, r, array *match*.
Ensure: Motifs u_1*gap of size d_1*u_2*gap of size d_2*u_3*$\dots u_m$

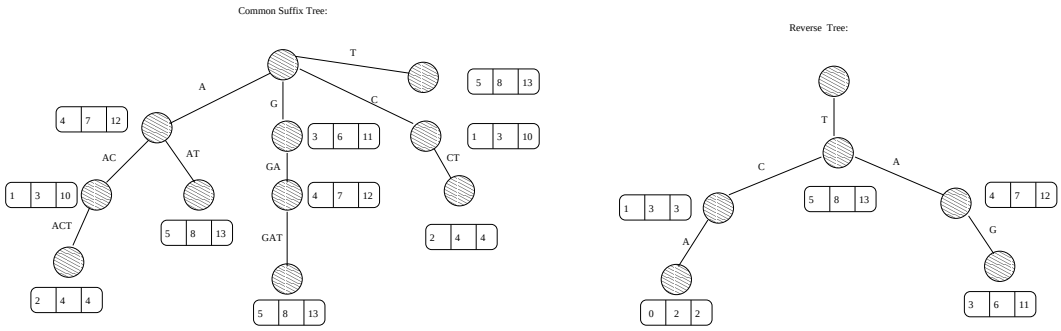
```
1: while L! = empty do
2:    u = pop(L);
3:    while M! = empty do
4:       v = (M) → top
5:       for i = 0 to r do
6:          if B_v[i] > B_u[i] then
7:             match++;
8:          end if
9:       end for
10:      if match == r then
11:         motif = Addnode(u, v)
12:         if |motif| ≤ q then
13:            u = motif
14:         end if
15:      end if
16:      v = v → next
17:   end while
18:   printmotif(motif);
19: end while
```

Once there are no more nodes in M that satisfy the condition of having their elements in their B array larger than the current array $B(u)$, this procedure stops and we report the motif found. Subsequently, we repeat the whole procedure with the next node u in stack L.

As an example, let's consider a set of sequences S: $S_1 = ACTGAT$, $S_2 = GTACTTGAT$ and $S_3 = CCACTAGTCACGAT$. The preprocessing steps construct $T(S)$ (Fig. 3a), $(\overleftarrow{T(S)})$, (Fig. 3b) and stacks L and M, (Fig. 4).

We will show the process of the algorithm using the fourth element of stack L which has the longest label $l(u) = ACT$ and $B(u) = \{2, 4, 4\}$. The first node v in stack M, that has

all its $B(v)$ positions larger than the positions of $B(u)$, has label G and $B(v) = \{3, 6, 11\}$. We combine the two nodes and compute the gaps by computing $(B(v) - B(u) - 1)$.



(a) Common segments trie (b) Reverse common trie

Fig. 3. The common segments trie and the reverse common trie of the example for variable gaps

Stack L	Labels
4, 7, 12	A
1, 3, 10	AC
5, 8, 13	AT
2, 4, 4	ACT
3, 6, 11	G
4, 7, 12	GA
5, 8, 13	GAT
1, 3, 10	C
2, 4, 4	CT
5, 8, 13	T

(a)

Stack M	Labels
0, 2, 2	A
1, 3, 3	C
3, 6, 11	G
4, 7, 12	A
5, 8, 13	T

(b)

Fig. 4. Figure (a) presents the stack L. Figure (b) presents the M stack which contains the nodes of the reverse trie.

We continue with the array $B(u)$, now having values $B(u) = \{3, 6, 11\}$. We again search stack M for the next node v that has $B(v)$ with larger values than the array $B(u)$. This node v has $B(v) = \{4, 7, 12\}$ and label A. We add the nodes, and continue in this manner until there are no more elements in stack M with suitable values in their B arrays. The final

motif found in this iteration is:

S_1= A C T G A T
S_2= G T A C T T G A T
S_3= C C A C T A G T C A C G A T G

4 Time and Space Complexity

During the preprocessing steps of the algorithm we construct the common segments trie $T(S)$ and reverse segments trie ($\overleftarrow{T(S)}$). Both constructions are bounded by $O(n)$ time and space complexity where n is the cumulative size of the lengths of the r strings in set S. Next, we insert all nodes from both tries in two different stacks. The total time and space needed for this step is again bounded by $O(n)$.

Algorithm 1 iterates the first stack L from the first element until the last one. This iteration requires $O(n)$ time to complete. Next, we update the $B(u)$ arrays of each element of the stack L (lines 5-8). This process requires $O(nr)$ time, where r, is the number of strings in set S and also the length of the B arrays. The subsequent search in stack M requires, in the worst case, $O(m)$ time. Thus, the complexity becomes $O(nrm)$. When we find matches between the nodes of L and M, we update the node u of L and repeat the search in M for a next match. So, there is an additional time requirement analogous to the matches found , k, and when $k > 0$, the total time complexity of this algorithm becomes $O(nrm + km)$

In algorithm 2, we pop each value from stack L until it is empty. Every time a node u is popped, we search in stack M for a node v that has larger elements in its $B(v)$ array than the array $B(u)$. This search requires $O(nr)$ time. Once we find a match, we combine the nodes u and v and then look at the stack M again, to find the next appropriate node v. So, we repeat this search in stack M, as many times as we find a match. So, the time complexity of this algorithm is $O(nrm + km)$,where k is the number of matches.

5 Conclusion

We have presented algorithms for two different variants of the problem of finding motifs with gaps. The first one is for finding common motifs with fixed gaps and the second one is for finding common motifs with variable gaps. The approach we suggest uses the structures of a common trie and a reverse trie to find the right and left part of the repeats comprising the motifs.

References

1. A. Apostolico. *The myriad virtues of subword trees in Combinatorial Algorithms on Words*. NATO ASI Series F12,Springer-Verlag, 1985.
2. A.M. Carvalho, Laurent Marsan, N. Pisanti, and Marie-France Sagot. Risotto: Fast extraction of motifs with mismatches. In *Proc. of the 7th Latin American Symposium on Theoretical Informatics (LATIN'06)*, pages 757–768, Valdivia, Chile, 2006.
3. R. Casadio and G. Myers, editors. *Algorithms in Bioinformatics, 5th International Workshop, WABI 2005, Mallorca, Spain, October 3-6, 2005, Proceedings*, volume 3692 of *Lecture Notes in Computer Science*. Springer, 2005.
4. C. Charras and T. Lecroq. *Exact string matching algorithms*. 2004.
5. T. Crawford, C. S. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology*, 11:73–100, 1998.
6. M. Crochemore, R. Grossi, N. Pisanti, and M.F Sagot. Bases of motifs for generating repeated patterns with wild cards. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 2(1):40–50, 2005.
7. M. Crochemore, C. S. Iliopoulos, M. Mohamed, and M.F. Sagot. Longest repeats with a block of don't cares. In *Proc. of the Latin American Theoretical Informatics (LATIN'04)*, 2004.
8. M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, Inc., New York, NY, USA, 1994.
9. A. Freitas, A. Carvalho, A. Oliveira, and M. Sagot. A highly scalable algorithm for the extraction of cis-regulatory regions, 2005.
10. D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA, 1997.
11. C. S. Iliopoulos, J. McHugh, P. Peterlongo, N. Pisanti, W. Rytter, and M. F. Sagot. A first approach to finding common motifs with gaps. *International Journal of Foundations of Computer Science*, 2004.
12. H. C. M. Leung, F. Y. L. Chin, R. Rosenfeld, and W.W. Tsang. Finding motifs with insufficient number of strong binding sites. *Journal of Computational Biology*, 12(6):686–701, 2005.
13. L.Marsan and M.F Sagot. Extracting structured motifs using a suffix tree - algorithms and application to promoter consensus identification. In *RECOMB '00: Proceedings of the fourth annual international conference on Computational molecular biology*, pages 210–219, New York, NY, USA, 2000. ACM Press.
14. D.W. Mount. *Bioinformatics : sequence and genome analysis*. Cold Spring Harbor Laboratory Press, 2001. Mount.
15. M.F. Sagot. Spelling approximate repeated or common motifs using a suffix tree. In *LATIN '98: Proceedings of the Third Latin American Symposium on Theoretical Informatics*, pages 374–390, London, UK, 1998. Springer-Verlag.