Motifs in sequences: localization and extraction

Maxime Crochemore* and Marie-France Sagot†

November 3, 2000

Contents

1 Motifs in sequences

Conserved patterns of any kind are of great interest in biology as they are likely to represent objects upon which strong constraints are potentially acting and

*`Maxime.Crochemore@univ-mlv.fr`, `http://www-igm.univ-mlv.fr/~mac`, Institut Gaspard-Monge, University of Marne-la-Vallée, F-77454 Marne-la-Vallée CEDEX 2.

†`Marie-France.Sagot@pasteur.fr`, `http://www-igm.univ-mlv.fr/~sagot`, Institut Pasteur, 25–28, rue du Docteur Roux, F-75724 Paris CEDEX 15, and Institut Gaspard-Monge.

may therefore perform a biological function. Among the objects which may model biological entities, we shall consider in this chapter strings only. As is by now well known, biological sequences, whether DNA, RNA or proteins, may be represented as strings over an alphabet of 4 letters (DNA/RNA) or 20 letters (proteins). Some of the basic problems encountered in classical text analysis have their counterpart when the texts are biological sequences, among them is pattern matching. However, this problem comes with a twist once we are in the realm of biology: exact patterns hardly make sense in this case. By *exact* above, we mean *identical*; and there are in fact at least two types of "non-identical" patterns one must consider in biology. One comes from looking at what "hides" behind each letter of the DNA/RNA or protein alphabet while the other corresponds to the more familiar notion of "errors". The errors concern mutational events which may affect a molecule during DNA replication. Those of interest to us are *point mutations*, that is, mutations operating each time on single letters of a biological sequence: *substitution*, *insertion* or *deletion*. Considering substitutions only is sometimes enough for dealing with some problems.

There are basically two questions that may be addressed when trying to search for known or predicted patterns in any text. Both are discussed in general computational biology books such as Durbin's *et al.* [1], Gusfield's [2], Meidanis and Setubal's [3] or Waterman's [4]. One, rather ancillary, is the question of position: where are these patterns localized (pattern localization prediction) ? The second question, more conceptual, concerns identifying and modeling the patterns *ab initio*: what would be a consensual *motif* for them (pattern consensus prediction) ? In biology, it is often the second question which is the most interesting although the first is far from being either trivial or solved. Indeed, in general what is interesting to discover is which patterns, unknown at start, match the string(s) more often than "expected" and have therefore a "chance" of representing an interesting biological entity. This entity may correspond to a binding site, *i.e.* to a (in general small) part of a molecule that will interact with another, or it may represent an element that is repeated in a dispersed or periodic fashion (for instance, tandemly). The role played by a repetition of whatever type is often unknown: some repeats, in particular small tandem ones, have been implicated in a number of genetic diseases and are also interesting for the purposes of studying polymorphism; other types of repeats, such as short inverted ones, seem to be hotspots for recombination.

We address both kinds of problems (pattern localization prediction and pattern consensus prediction) after having discussed some notions of "non-identity", that is, of similarity, that we shall be considering. These are presented in Section 2. We start with the identity, both because it may sometimes be of interest and because this allows us to introduce some notations that are used throughout the paper. Such notations are based on those adopted by Karp *et al.* in a pioneering paper on finding dispersed exact repeats in a string [5]. From there, it is easy to derive a definition of similarity based, not on the identity, but on any relation between the letters of the alphabet for the strings. In particular, this relation can be, and in general is, non transitive (contrary to equality). This was introduced by Soldano *et al.* [6]. Finally, definitions of similarity taking errors (substitutions, insertions and deletions) into account are discussed and the idea of models is presented. This idea was initially formally defined by Sagot *et al.* [7].

We review the pattern localization prediction question in Section 3. Since many methods used to locate patterns are inspired from algorithms developed for matching fixed patterns with equality, we state the main results concerning this problem. Complexity bounds have been intensively studied and are known with a good accuracy. This is the background for broader methods aimed at locating approximate patterns. The most widely used approximation is based on the three alignment operations recalled in Section 2. The general method designed to match an approximate pattern is an extension of the dynamic programming method used for aligning strings. Improving this method has also been intensively investigated because of the multitude of applications it generates. The fastest known algorithms are for a specialization of the problem with weak but extra conditions on the scores of edit operations.

For fixed texts, pattern matching is more efficiently solved by using some kind of index. Indexes are classical data structures aimed at providing a fast access to textual databases. As such, they can be considered as abstract data types or objects. They consist both of data structures to store useful information and of operations on the data (see Salton [8], or Baeza-Yates and Ribero-Neto [9]). The structures often memorize a set of keys as is the case of an index at the end of a technical book. Selecting keys is a difficult question that sometimes requires human action. In the chapter, we consider full indexes, which contain all possible factors (segments) of the original text, and we refer to these structures as factor or suffix structures. These structures help finding repetitions in strings, search for other regularities, solve approximate matchings, or even match two-dimensional patterns, to quote a few applications. Additional or deeper analysis of pattern matching problems may be found in books by Apostolico and Galil [10], Crochemore and Rytter [11], Gusfield [2], and Stephen [12].

Section 4 deals with the problem of finding repeats, exact or approximate, dispersed or appearing in a regular fashion along a string. Perhaps the most interesting work as concerns this area is that of Karp *et al.* [5] for identifying exact, dispersed repeats. This is discussed in some detail. Combinatorial algorithms also exist for finding tandem repeats. The most interesting ones are due to Landau [13] and Kannan and Myers [14], which allows for any error scoring system, and to Kurtz *et al.* [15], which uses a suffix tree for locating such repeats and comes with a very convenient visualisation tool. In biology, so called *satellites* constitute another important type of repetitions. Satellites are tandem arrays of approximate repeats varying in the number of occurrences between two and a few millions and in length between two and a few hundreds, sometimes thousands of letters. Only one combinatorial formulation of the problem has been given to this date [16], which we describe at some length.

Finally, motif extraction is considered in Section 5. A lot of the initial work done in this area used a definition of similarity that is based on the relative entropy of the occurrences of a motif in the considered set of strings. This produces often good results for relatively small data-sets, and the method has therefore being continuously improved. Such a definition, however, leads to exact algorithms that are exponential in the number of strings and heuristics have therefore to be employed. These do not guarantee optimality, that is, they do not guarantee that the set of occurrences given as a final solution is the one having maximal relative entropy. We do not treat such methods in the chapter. The author is referred to [73] for a survey of these and other methods from the point of view of biology.

A definition of similarity based on the idea of models (which are objects that are external to the strings) and of a maximum error rate between such models and their occurrences in strings can lead to combinatorial algorithms. Some algorithms in this category are efficient enough to be used for more complex models. An algorithm for extracting simple models as well as more complex ones, called *structured models*, elaborated by Marsan *et al.* [17] is treated in some detail.

2 Notions of similarity

2.1 Preliminary definitions

If s is a string of length $|s| = n$ over an alphabet Σ, that is, $s \in \Sigma^n$, its individual elements are noted s_i for $1 \leq i \leq n$, so that we have $s = s_1 s_2 \ldots s_n$. A non empty word $u \in \Sigma^*$ is a *factor* of s if $u = s_i s_{i+1} \ldots s_j$ for a given pair (i, j) such that $1 \leq i \leq j \leq n$. The empty word, denoted by λ, is also a factor of s.

2.2 Identity

Although identity is seldom an appropriate notion of similarity to consider when working with biological objects, it may sometimes be of interest. This is a straightforward notion we nevertheless define properly as this allows us to introduce some notations that is used throughout the paper.

The identity concerns words in a string and we therefore adopt Karp *et al.* [5] identification of such words by their start position in the string. To facilitate exposition, this and all other notions of similarity are given for words inside a single string. It is straightforward to adapt them to the case of more than one string (for instance, by considering the string resulting from the concatenation of the initial ones with a distinct forbidden symbol separating any two adjacent strings). Let us note E the identity relation on the alphabet Σ (the E stands for "Equivalence").

Relation E between elements of Σ may then be extended to a relation E_k between factors of length k in a string s in the following way:

Definition 2.1 *Given a string $s \in \Sigma^n$ and i, j two positions in s such that $i, j \leq n - k + 1$, then:*

$$i \; E_k \; j \Leftrightarrow s_{i+l} \; E \; s_{j+l} \textit{ for all } l \textit{ such that } 0 \leq l \leq (k-1).$$

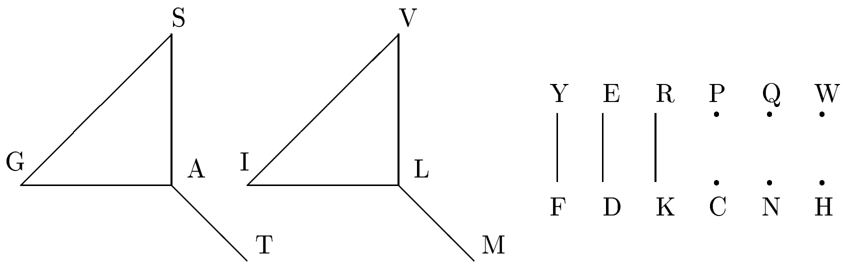


Figure 1: Example of a relation of similarity between the letters of the protein alphabet (called amino acids).

In other words, iE_kj if and only if $s_i s_{i+1} \dots s_{i+k-1} = s_j s_{j+1} \dots s_{j+k-1}$. For each $k \geq 1$, E_k establishes an equivalence relation that corresponds to a relation between occurrences of words of length k in s. This provides a first definition of similarity between such occurrences. Indeed, each equivalence class of E_k having cardinality greater than one is the witness of a repetition in s.

2.3 Non transitive relation

When dealing with biological strings, one has to consider that the "letters" represented by such strings are complex biological objects with physico-chemical properties, as, for instance, electrical charge, polarity, size, different levels of acidity, *etc.* Some, but seldom all, of these properties may be shared by two or more objects. This applies more to proteins than to DNA/RNA but is true to some extent for both.

A more realistic relation to establish between the letters of the protein or DNA/RNA alphabet (respectively called amino acids and nucleotides) would therefore be reflexive, symmetric but non transitive [6]. An example of such a relation, noted R, is given below.

Example 1 *Let* $\Sigma = \{A,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,V,W,Y\}$ *be the alphabet of amino acids and* R *be the relation of similarity between these amino acids given by the graph given in Figure 1. The maximal cliques of* R *are the sets:* $\{A,S,G\}$, $\{A,T\}$, $\{I,L,V\}$, $\{L,M\}$, $\{F,Y\}$, $\{D,E\}$, $\{K,R\}$, $\{C\}$, $\{P\}$, $\{N\}$, $\{Q\}$, $\{H\}$, $\{W\}$.

It may be represented by a graph whose nodes are the elements of Σ and where an edge links two nodes if the elements of Σ labeling the nodes correspond to biological objects sharing enough physico-chemical properties to be considered similar.

As previously, the relation R between elements of Σ may easily be extended to a relation R_k between factors of length k in a string s.

Definition 2.2 *Given a string* $s \in \Sigma^n$ *and* i, j *two positions in* s *such that* $i, j \leq n - k + 1$*, then:*

$$i \; R_k \; j \Leftrightarrow s_{i+l} \; R \; s_{j+l} \textit{ for all } l \textit{ such that } 0 \leq l \leq (k-1).$$

For each $k \geq 1$, R_k establishes a relation that is no longer an equivalence between positions (factors of length k) in a string s. The concept that is important here is that of a (maximal) clique.

Definition 2.3 *Given an alphabet Σ and a non transitive relation on Σ, a set C of elements of Σ is a (maximal) clique of relation R if for all $\alpha, \beta \in C$, $\alpha \; R \; \beta$ and for all $\gamma \in \Sigma \setminus C$, $C \bigcup \{\gamma\}$ is not a clique.*

Definition 2.4 *Given a string $s \in \Sigma^n$, a set C_k of positions in s is a clique of relation R_k if for all $i, j \in C_k$, $i \; R_k \; j$ and for all $l \in [1..n] \setminus C_k$, $C_k \bigcup \{l\}$ is not a clique.*

Cliques of R_k give us then a second way of establishing a definition of similarity between factors of length k in a string.

2.4 Allowing for errors

Introducing the idea of a model

Let us initially assume that the only authorized errors are substitutions. In view of the definitions established in previous sections, one would be tempted to define a relation of similarity H between two factors of length k in a string s, that is, between two positions i and j in s, the following way.

Definition 2.5 *Given a string $s \in \Sigma^n$ and i, j two positions in s such that $i, j \leq n-k+1$, then:*

$$i \; H_k \; j \Leftrightarrow dist_H(s_i \ldots s_{i+k-1}, \; s_{j \ldots s_{j+k-1}}) \leq e$$

where $dist_H(u, v)$ is the Hamming distance (hence the H) between u and v (that is, the minimum number of substitutions one has to operate on u in order to obtain v) and e is a non negative integer that is fixed.

Parameter e corresponds to the maximum number of substitutions that are tolerated. In the same way as in Section 2.3, cliques of H_k provide us with another possible definition of similarity between factors of length k in a string.

Eevn before trying to consider how to adapt the above definition to the case of a Levenshtein (or any other type of) distance where insertions and deletions are permitted besides substitutions (this is not completely trivial: indeed, given two words u and v respectively starting at positions i and j in s and such that iL_kj, what is the meaning of k ?), one may intuitively note that calculating H_k (and, *a fortiori*, L_k) is no longer as easy as computing E_k or R_k.

The reason is that, although the definitions given in Sections 2.2 and 2.3 involve pairs of positions in a string s, it is possible to rewrite them in such a way that, given a position i in s and a length k, it is immediate to determine to which class or clique(s) i belongs in the sense that the class or clique(s) can be uniquely identified just by "reading" $s_i \ldots s_{i+k-1}$. Let us consider first the simpler case of an identity. Straightforwardly, position i belongs to the class whose label is $s_i \ldots s_{i+k-1}$. In the case of a non transitive relation R between letters of Σ, let us name C the set of (maximal) cliques of R and note $clique_R(\alpha)$ the cliques of R to which a letter α belongs. Then, position i belongs to all the sets of R_k whose labels may be spelled from the (regular) expression $clique_R(s_i) \ldots clique_R(s_{i+k-1})$ and that are maximal under R_k. Note the small difference here with the identity relation: maximality of a validly labeled set has to be checked [6].

No such easy rewriting and verification are possible in the case of the definition of H_k (or L_k had we already written it) if we wish to build the notion of similarity between factors in a string upon that of the cliques of H_k. Indeed, obtaining such cliques needs comparing (a possibly great number of) pairs of positions between themselves. This is expensive.

One may, however, rewrite the definition of H_k in a way that refers to labels as we did above for E_k and R_k although such labels are no longer as immediately identifiable. A possible definition (still for the case where substitutions only are considered) is the following.

Definition 2.6 *Given a string $s \in \Sigma^n$ and i, j two different positions in s such that $i, j \leq n - k + 1$, then:*

$$i \; H_k \; j \Leftrightarrow \exists m \in \Sigma^k \text{ such that } dist_H(m, s_i \ldots s_{i+k-1}) \leq e \text{ and } dist_H(m, s_j \ldots s_{j+k-1}) \leq e$$

where $dist_H(u, v)$ and e are as before.

Generalizing this, gives the following definition.

Definition 2.7 *A set S_k of positions in s represents a set of factors in s of length k that are all similar between themselves if, and only if, there exists (at least) a string $m \in \Sigma^k$ such that, for all elements i in S_k, $dist_H(m, s_i \ldots s_{i+k-1}) \leq e$ and, for all $j \in [1..n] \setminus S_k$, $dist_H(m, s_i \ldots s_{i+k-1}) > e$.*

Observe that extension of both definitions to a Levenshtein distance becomes now straightforward. We reproduce below, after modification, just the last definition.

Definition 2.8 *A set S_k of positions in s represents a set of factors of length k that are similar if, and only if, there exists (at least) a string $m \in \Sigma^k$ such that, for all elements i in S_k, $dist_L(m, s_i \ldots) \leq e$ and, for all $j \in [1..n] \setminus S_k$, $dist_L(m, s_i \ldots) > e$.*

Since the length of an occurrence of a model m may now be different from that of m itself (it varies between $|m| - e$ and $|m| + e$) we denote the occurrence by $(s_i \ldots)$ leaving indefinite its right-end point.

Observe also that it remains possible, given a position i in s and a length k, to obtain the label of the group(s) of the relation H_k (or L_k) i belongs to. Such labels are represented by all strings $m \in \Sigma^k$ such that $dist_H$(or $dist_L$)$(m,s_i \ldots)$ $\leq e$, that is, such that their distance from the word starting at position i in s is no more than e.

We call *models* such group labels. Positions in s indicating the start of a factor of length k are *e-occurrences* (or simply *occurrences* where there is no ambiguity) of a model m if $dist(m,s_i \ldots) \leq e$ where $dist$ is either the Hamming or Levenshtein distance. Observe that a model m may have no exact occurrence in s.

Finally, we have considered so far what is called a "unitary cost distance" (unitary because the cost of each operation, substitution, insertion or deletion, is one unit). We could have used instead a "weighted cost distance", that is, we could have used any cost for each operation, in the range of integers or real numbers.

Expanding on the idea of models – Two more possible definitions of similarity

Non transitive relation and errors Models allow us to considerably enrich the notion of conservation. For instance, it enables us to simultaneously consider a non relative transition between the letters of the alphabet (amino acids or nucleotides) and the possibility of errors. In order to do that, it suffices to permit the model to be written over an extended alphabet composed of a subset of the set of all subsets of Σ (noted $\mathcal{P}(\Sigma)$) where Σ is the alphabet of amino acids or nucleotides. Such an alphabet can be, for instance, one defined by the maximal cliques of the relation R given in Figure 1. Definition 2.8 of Section 2.4 then becomes:

Definition 2.9 *A set S_k of positions in s represents a set of factors of length k that are all similar between themselves if, and only if, there exists (at least) one element $M \in P^k$ with $P \subseteq \mathcal{P}(\Sigma)$ such that, for all elements i in S_k, $setdist(M,s_i \ldots) \leq e$ and, for all $j \in [1..n] \setminus S_k$, $setdist(M,s_i \ldots) > e$, where $setdist(M,v)$ for $M \in P$ and $u \in \Sigma$ is the minimum Hamming or Levenshtein distance between v and all $u \in M$.*

Among the subsets allowed in P, the alphabet of models, may be $\{\Sigma\}$ itself, that is the *wild card*. It is obvious that this may lead to trivial models. Alphabet P may then come with weights attached to each of its elements indicating how many times (possibly infinite) it may appear in an interesting model. Observe that another way of describing the alphabet P of models is as the set of edges of a (possibly weighted) hypergraph whose nodes are the elements of Σ.

When e is zero, we obtain a definition of similarity between factors in the string that closely resembles that given in Section 2.3. Note however that, given two models M_1 and M_2, we may well have that the set of occurrences of M_1 is included in that of M_2. The cliques of Definition 2.4 correspond to the sets of occurrences that are maximal.

A word instead of symbol-based similarity Errors between a group of similar words and the model of which they are occurrences can either be counted as unitary events (possibly with different weights) as was done in the previous sections, or they can be given a score. The main idea behind scoring a resemblance between two objects is that it allows to average the differences that may exist between them. It may thus provide a more flexible function for measuring the similarity between words. A simple example illustrates this point.

Example 2 *Let* $\Sigma = \{A,B,C\}$ *and:*

score(i,i)	$=$ *1,*		$\forall\ i \in \Sigma$*;*
score(A,B)	$=$ *score(B,A)*	$=$ *-1;*	
score(A,C)	$=$ *score(C,A)*	$=$ *-1;*	
score(B,C)	$=$ *score(C,B)*	$=$ *-1.*	

If we say that 2 words are similar either if:

- *the number of substitutions between them is* $\leq$ *1,*
- *their score is* $\geq$ *1,*

then by the first criterion the words $AABAB$ *and* $AACCB$ *are not similar, while by the second criterion they are, the second substitution being allowed because the two words on the average share enough resemblance.*

In the example and in the definition of similarity introduced in this section, gaps are not allowed, only substitutions are. This is done essentially for the sake of clarity. Gaps may, however, be authorized, the reader is referred to [18] for details.

Let a numerical matrix $\mathcal{M}$ of size $\mid \Sigma \mid \times \mid \Sigma \mid$ be given such that:

$$\mathcal{M}(a,b) = \text{score between } a \text{ and } b \text{ for all } a,\, b \in \Sigma.$$

If this score measures a similarity between a and b, we talk of a similarity matrix (two well-known examples of which in biology are PAM250 [19] and BLOSUM62 [20]), while if the score measures a dissimilarity between a and b we talk of a dissimilarity matrix. A special case of this latter matrix is when the dissimilarity measure is a metric, that is when the scores obey, among other conditions, the triangular inequality. In that situation, we talk of a distance matrix (an example of which is the matrix proposed by J.-L. Risler [21]).

In what follows, we consider that $\mathcal{M}$ is a similarity matrix.

Definition 2.10 *Given* $u = u_1u_2...u_k \in \Sigma^k$, $m = m_1m_2...m_k \in \Sigma^k$ *a model of length* k *and* $\mathcal{M}$ *a matrix, we note:*

$$score_{\mathcal{M}}(u,m) = \sum_{i=1}^{k} \mathcal{M}(u_i, m_i).$$

Definition 2.11 *A set* S_k *of positions in* s *represents a set of factors of length* k *that are similar if, and only if, given* w *a positive integer such that* $w \leq k$ *and* t *a threshold value:*

1. *there exists (at least) one element* $m \in \Sigma^k$ *such that, for all elements* i *in* S_k *and for all* $j \in \{1,\ldots,\mid m \mid -w+1\}$, $score_{\mathcal{M}}(m_j \ldots m_{j+w-1}, s_i \ldots s_{i+w-1}) \geq t$;
2. *for all* $i \in [1..n] \setminus S_k$, *there exists at least one* $j \in \{1,\ldots,\mid m \mid -w+1\}$ *such that* $score_{\mathcal{M}}(m_j \ldots m_{j+w-1}, s_i \ldots s_{i+w-1}) < t$.

An example is given below.

Example 3 *Let* $\Sigma = \{A,B,C\}$, $w = 3$ *and* $t = 6$. *Let* $\mathcal{M}$ *be the following matrix:*

	A	*B*	*C*
A	*3*	*1*	*0*
B	*1*	*2*	*1*
C	*0*	*1*	*3*

Given the three strings:

$s1 = ABCBBABB\underline{BACABACB}BBBAB$

$s2 = \underline{CABACAAC}BACCABCACCACCC$

$s3 = BBBACACCABA\underline{BACABACA}BA$

then the longest model that is present in all strings is CACACACC (at positions 9, 1 and 12 respectively).

3 Motif localization

We review in this section the main results and combinatorial methods used to locate patterns in strings. The problem is of main importance for several reasons. From a theoretical point of view, it is a paradigm for the design of efficient algorithms. From a practical point of view, the algorithms developed in this chapter often serve as basic components in string facility software. In particular, some techniques are used for the extraction of unknown motifs.

We consider two instances of the question, depending on whether the motif is fixed or the string is fixed. In the first case, preprocessing the pattern accelerates the search for it in any string. Searching on a fixed string is made faster if a kind of index on it is preprocessed. At the end of the section, we sketch how to search structural motifs for the identification of tRNAmotifs in biological sequences.

3.1 Searching for a fixed motif

String searching or string matching is the problem of locating all the occurrences of a string x of length p, called the pattern, in another string s of length n, called the sequence or the text. The algorithmic complexity of the problem is analyzed by means of standard measures: running time and amount of memory space required by the computations. This section deals with solutions in which the pattern is assumed to be fixed. There are mainly three kinds of methods to solve the problem: sequential methods (simulating a finite automaton), practically-fast methods, and time-space optimal methods. Methods that search for occurrences of approximate patterns are discussed in the next subsection. Alternative solutions based on a preprocessing of the text are described in a following subsection.

Efficient algorithms for the problem have a running time that is linear in the size of the input (*i.e.* $O(n+p)$). Most algorithms require an additional amount of memory space that is linear in the size of the pattern (i.e. $O(p)$). Information stored in this space is computed during the preprocessing phase, and later used during the search phase. The time spent during the search phase is particularly important. The number of comparisons made and the number of inspections executed have therefore been evaluated with great care. For most algorithms, the maximum number of comparisons (or number of inspections) made during the execution of the search is less than $2n$. The minimum number of comparison necessary is $\lfloor n/p \rfloor$, and some algorithms reach that bound in ideal situations.

The complexity of the string searching problem is given by the following theorem due to Galil and Seiferas (1983). The proof is based on space-economical methods that are outside the scope of this chapter (see [11], for example). Linear time is however met by many other algorithms. Note that in the "O" notation, coefficients are independent of the alphabet size.

Theorem 1 *The string searching problem, locating all occurrences of a pattern x in a text s, can be solved in linear time, $O(|s|+|x|)$, with a constant amount of additional memory space.*

The average running time of the search phase is sometimes considered as more significant than the worst-case time complexity. Despite the fact that it is usually difficult to model the probability distribution of specific texts, results for a few algorithms (with a hypothesis on what "average" means) are known. Equiprobability of symbols and independence between their occurrences in texts represent a common hypothesis used in this context and gives the next result (Yao, 1979). Although the hypothesis is too strong, the result reflects the actual running time of algorithms based on the method described below. In addition, it is rather simple to design a string searching algorithm working in this time span.

Theorem 2 *Searching a text of length n for a preprocessed pattern of length p can be done in optimal expected time $O(\frac{\log p}{p} \times n)$.*

String searching algorithms can be classified into three classes. In the first class, the text is searched sequentially, one symbol at a time from beginning to end. Thus all symbols of the text (except perhaps $p-1$ of them at the end) are inspected. Algorithms simulate a recognition process using a finite automaton. The second class contains algorithms that are practically fast. The time complexity of the search phase can even be sublinear, under the assumption that both the text and the pattern reside in main memory. Algorithms from the first two classes usually require $O(p)$ extra memory space to work. Algorithms from the third class show that the additional space can be reduced to a few integers stored in a constant amount of memory space. Their interest is mainly theoretical so far.

The above classification can be somehow refined by considering the way the search phases of algorithms are designed. It is convenient to consider that the text is examined through a *window*. The window is assimilated to the segment of the text it contains and it has usually the length of the pattern. It runs along the text from beginning to end. This scheme is called the *sliding window* strategy and is described below. It uses a scan-and-shift mechanism.

1. put window at the beginning of text;
2. **while** window on text **do**
3. *scan*: **if** window = pattern **then** report it;
4. *shift*: shift window to the right and
5. memorize some information for use during next scans and shifts;

During the search, the window on the text is periodically shifted to the right according to rules that are specific to each algorithm. When the window is placed at a certain position on the text, the algorithm checks whether the pattern occurs there, *i.e.*, if the pattern equals the content of the window. This is the *scan* operation during which the algorithm acquires from the text information that is often used to determine the next shift of the window. Part of the information can also be kept in memory after the shift operation. This information is then used for two purposes: first, saving time during the next scan operations, and, second, increasing the length of further shifts. Thus, the algorithms operate a series of alternate scans and shifts.

A naive implementation of the scan-and-shift scheme (no memorization, and uniform shift of length 1) leads to a searching algorithm running in maximum time $O(p \times n)$; the expected number of comparisons is $4n/3$ on a four-letter alphabet. This performance is quite poor as compared to preceding results.

Practically fast searches

We describe a string searching strategy that is considered as the fastest in practice. Derived algorithms apply when both the text and the pattern reside in main memory. We thus do not take into account the time to read them. Under this assumption, some algorithms have a sublinear behavior. The common feature of these algorithms is that they scan the window in the reverse direction (from right to left).

The classical string searching algorithm that scans the window in reverse direction is the BM algorithm (Boyer and Moore, 1977). At a given position in the text, the algorithm first identifies the longest common suffix u of the window and the pattern. A match is reported if it equals the pattern. After that, the algorithm shifts the window to the right. Shifts are done in such a way that the occurrence of u in the text remains aligned with an equal segment of the pattern, and are often called *match shifts*. The length of the shift is determined by what is called the *displacement* of u inside x, and denoted by $d(u)$. A sketch of the BM algorithm is displayed below.

```
while window on text do
    u := longest common suffix of window and pattern;
    if u = pattern then report a match;
    shift window d(u) places to the right;
```

The function d depends only on the pattern x so that it can be precomputed before the search starts. In the BM algorithm, an additional heuristics on mismatch symbols of the text is also usually used. This yields another displacement function used in conjunction with d. It is a general method that may improve almost all algorithms in certain real situations.

The BM algorithm is memoryless in the sense that, after a shift, it starts scanning the window from scratch. No information about previous matches is kept in memory. When the algorithm is applied to find all occurrences of A^p inside A^n, the search time becomes proportional to $p \times n$. The reason for the quadratic behavior is that no memory is used at all. It is, however, very surprising that BM algorithm turns out to be linear when the search is limited to the first occurrence of the pattern. By the way, the original algorithm has been designed for that purpose. Only very periodic patterns may increase the search time to a quadratic quantity, as shown by the next theorem (Cole, 1990). The bound it gives is the best possible. Only a modified version of the BM algorithm can therefore make less than $2n$ symbol comparisons at search time.

Theorem 3 *Assume that pattern x satisfies period$(x) > |x|/2$. Then, the BM searching algorithm performs at most $3|s| - |s|/|x|$ symbol comparisons.*

The theorem also suggests that only little information about configurations encountered during the process has to be kept in memory in order to get a linear time search for any kind of patterns. This is achieved, for instance, if prefix memorization is performed each time an occurrence of the pattern is found. However, this is also achieved with a better bound by an algorithm called TURBO_BM. This modification of the BM algorithm forgets all the history of the search, except for the most recent one. Analysis becomes simpler, and the maximum number of comparisons at search phase becomes less than $2n$.

Searching simultaneously for several (a finite number of) patterns can be done more efficiently than searching for them one at a time. The natural procedure takes an automaton as pattern. It is an extension of the single-pattern searching algorithms based on the simulation of an automaton. The standard solution is from Aho and Corasick [22].

3.2 Approximate matchings

The search for approximate matchings of a fixed pattern produces the position in the text s of an approximation of the pattern x. Searching texts for approximate matchings is usually done by methods derived from the exact string searching problem described above. They either include an exact string matching as an internal procedure or they transcribe a corresponding algorithm. The two classical ways to model approximate patterns consist in assuming that a special symbol can match any other symbol, or that operations to transform a pattern into another are possible.

In the first instance we have, in addition to the symbols of the input alphabet Σ, a wild card (also called a *don't care* symbol) ϕ with the property that ϕ matches any other character in Σ. This gives rise to variants of the string searching problem where, in principle, ϕ appears (i) only in the pattern, (ii) only in the text, or (iii) both in the pattern and the text. Variant (i) is solved by an adaptation of the multiple string matching and of the pattern-matching automaton of Aho and Corasick [22]. For other variants, a landmark solution is by Fischer and Paterson [23]. They transpose the string searching problem into an integer multiplication problem, thereby obtaining a number of interesting algorithms. This observation brings string searching into the family of boolean, polynomial and integer multiplication problems and leads to an $O(n \log p \log \log p)$ time solution in the presence of wild cards (provided that the size of Σ is fixed).

The central notion for comparing strings is based on three basic *edit operations* on strings introduced in Section 2. It may be assumed that each edit operation has an associated nonnegative real number representing the *cost* of that operation, so that the cost of deleting from w an occurrence of symbol a is denoted by $D(a)$, the cost of inserting some symbol a between any two consecutive positions of w is denoted by $I(a)$ and the cost of substituting some occurrence of a in w with an occurrence of b is denoted by $S(a, b)$.

The *string editing problem* for input strings x and s consists in finding a sequence of edit operations, or *edit script*, Γ of minimum cost that transforms x into s. The cost of Γ is the *edit distance* between x and s (it is a mathematical distance under some extra hypotheses on operation costs). Edit distances where individual operations are assigned unit costs occupy a special place.

It is not difficult to see that the general problem of edit distance computation can be solved by an algorithm running in $O(p \times n)$ time and space through dynamic programming. Due to the widespread application of the problem, however, such a solution and a few basic variants were discovered and published in an extensive literature. The reader can refer to Apostolico and Giancarlo (1998) [24], or to [10] for a deeper exposition of the question.

The computation of edit distances by dynamic programming is readily set up. For this, let $C(i, j)$ ($0 \leq i \leq |s|$ and $0 \leq j \leq |x|$) be the minimum cost of transforming the prefix of s of length i into the prefix of x of length j. Then $C(0, 0) = 0$, $C(i, 0) = C(i-1, 0) + D(s_i)$ ($i = 1, 2, ..., |s|$), $C(0, j) = C(0, j-1) + I(x_j)$ ($j = 1, 2, ..., |x|$), and $C(i, j)$ equals

$$\min\{C(i-1, j-1) + S(s_i, x_j),\ C(i-1, j) + D(s_i),\ C(i, j-1) + I(x_j)\}$$

for all i, j, ($1 \leq i \leq |s|$, $1 \leq j \leq |x|$). Observe that, of all entries of the C-matrix, only the three entries $C(i-1, j-1)$, $C(i-1, j)$, and $C(i, j-1)$ are involved in the computation of the final value of $C(i, j)$. Hence $C(i, j)$ can be evaluated row-by-row or column-by-column in $\Theta(|s| \times |x|) = \Theta(p \times n)$ time. An optimal edit script can be retrieved at the end by backtracking through the local decisions made by the algorithm.

A few important problems are special cases of string editing, including the computation of a *longest common subsequence*, *local alignment*, *i.e.*, the detection of local similarities in strings, and some important variants of *string searching with errors*, or searching for occurrences of approximate patterns in texts.

String Searching with differences

Consider the problem of computing, for every position of the textstring s, the best edit distance achievable between x and a substring w of s ending at that position. Under the unit cost criterion, a solution is readily derived from the recurrence for string editing given above. The first obvious change consists in setting all costs to 1 except that $S(x_i, s_j) = 0$ for $x_i = s_j$. We thus have now, for all i, j, $(1 \leq i \leq |x|, 1 \leq j \leq |s|)$,

$$S(i,j) = \min\{S(i-1,j-1)+1,\ S(i-1,j)+1,\ S(i,j-1)+1\}.$$

A second change affects the initial conditions, so that we have now $S(0,0) = 0$, $S(i,0) = i$ $(i = 1,2,...,p)$, $S(0,j) = 0$ $(j = 1,2,...,n)$. This has the effect of setting to zero the cost of prefixing x by any prefix of s. In other words, any prefix of the text can be skipped at no cost in an optimum edit script.

The computation of S is then performed in much the same way as indicated in table C above, thus taking $\Theta(|x| \times |s|) = \Theta(p \times n)$ time. We are interested now in the entire last row of matrix S.

In practice, it is often more interesting to locate only those segments of s that present a high similarity with x under the adopted measure. Formally, given a pattern x, a text s, and an integer e, this restricted version of the problem consists in locating all terminal positions of substrings w of s such that the edit distance between w and x is at most e. The recurrence given above will clearly produce this information. However, there are more efficient methods to deal with this restricted case. In fact, a time complexity $O(e \times n)$ and even sublinear expected time are achievable. We refer to, *e.g.*, [10, 11] for detailed discussions. In the following, we review some of the basic principles behind an $O(e \times n)$ algorithm for string searching with e differences due to Landau and Vishkin (1986). Note that when e is a constant the corresponding time complexity then becomes linear.

It is essential here that edit operations have unitary costs. Matrix S has an interesting property that is intensively used to get the $O(e \times n)$ running time: its values are in increasing order along diagonals, and consecutive values on a same line or a same column differ by at most one unit (see Figure 2). Because of the monotonicity property on diagonals and unitary costs, the interesting positions on diagonals are those corresponding to a strict incrementation. Computing these values only produces a fast computation in time $O(e \times n)$. This is possible if queries on longest common prefixes, as suggested in Figure 2, are answered in constant time. This, in turn, is possible because strings can be preprocessed in order to get this time bound.

R		-1	0	1	2	3	4	5	6	7	8	9	10	11
			`C`	`A`	`G`	`A`	`T`	`A`	`A`	`G`	`A`	`G`	`A`	`A`
-1		**0**	**0**	0	**0**	**0**	**0**	**0**	0	**0**				
0	`G`	**1**	1	1	0	1	1	1	1	0				
1	`A`			**1**	1	0	1	**1**	**1**	1	**0**			
2	`T`					1	0	1			**1**	1		
3	`A`						1	0	1				1	
4	`A`							**1**	**0**	**1**				**1**

Figure 2: Simulation of fast searching for approximate matchings. Searching y = `CAGATAAGAGAA` for x = `GATAA` with at most one difference. Pattern x occurs at right positions 6 on y without errors (since $R[4,6] = 0$), and at right positions 5, 7 et 11 with one error (since $R[4,5] = R[4,7] = R[4,11] = 1$). After initialization, values are computed diagonalwise, value 0 during the first step and value 1 during the second step. Value $R[4,6] = 0$ comes from the fact that `GATAA` is the longest common prefix of x and $y[2\,..\,11]$. And, as a second example, $R[4,11] = 1$ because `AA` is the longest common prefix of $x[3\,..\,4]$ and $y[10\,..\,11]$. When queries related to longest common prefixes are answered in constant time the running time is proportional to bold values in the table.

To do so, we consider the suffix tree (see section 3.3 below), $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(z))$, of $z = x\$s$ where $\$ \notin \mathit{alph}(s)$. String $w = \mathit{LCP}(x[\ell+1\,..\,p-1], s[d+\ell+1\,..\,n-1])$ is also $\mathit{LCP}(x[\ell+1\,..\,p-1]\$s, s[d+\ell+1\,..\,n-1])$ because $\$ \notin \mathit{alph}(s)$. Let f and g be the nodes of $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(z))$ associated with strings $x[\ell+1\,..\,p-1]\$s$ and $s[d+\ell+1\,..\,n-1]$. Their common prefix of maximal length is then the label of the path in the suffix tree starting at the root and ending at the lowest common ancestor of f and g. Longest common prefix queries are thus transformed into lowest common ancestor queries that are answered in constant time by an algorithm due to Harel and Tarjan [25]. (1984), simplified latter by Schieber and Vishkin [26]. (1988). The consequence of the above discussion is the next theorem.

Theorem 4 *On a fixed alphabet, after preprocessing x and s, searching s for occurrences of x with at most e differences can be solved in time $O(e \times |s|)$.*

In applications to massive data, even a $O(e \times n)$ time may be prohibitive. By using filtration methods, it is possible to set up sublinear expected time queries. One possibility is to first look for regions with exact replicas of some pattern segment and then scrutinize those regions. Another possibility is to look for segments of the text that are within a small distance of some fixed segments of the pattern. Some of the current top performing software for molecular database searches are engineered around these ideas [27, 28, 29, 30]. A survey may be found in [31].

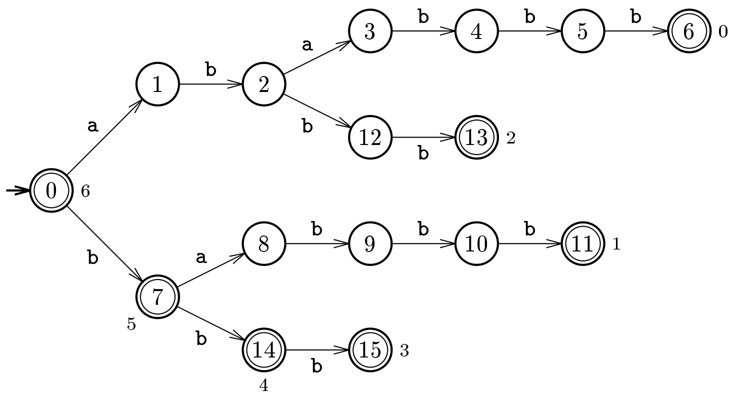


Figure 3: Suffix trie of ababbb.

3.3 Indexing

Full indexes are designed to solve the pattern matching problem, searching s for occurrences of x, when the text s is fixed. Having a static text allows to build a data structure to which the queries are applied. Efficient solutions require a preprocessing time $O(|s|)$ and need $O(|x|)$ searching time for each query.

Full indexes store the set of factors of the text s. Since factors are beginnings of suffixes of s, this is equivalent to storing all suffixes of the text. Basic operations on the index are: find if pattern x occurs in s, give the number of occurrences of x in s, and list all positions of these occurrences. But many other operations admit fast solutions through the use of indexes.

Indexes are commonly implemented by suffix trees, suffix automata (also called suffix DAWG's, Directed Acyclic Word Graphs), or suffix arrays. The latter structure realizes a binary search in the ordered list of suffixes of the text. The former structures are described in the remaining of the section.

Suffixes of s can be stored in a digital tree called the suffix trie of s. It is an automaton whose underlying graph is a tree. Branches are labeled by all the suffixes of s. More precisely, the automaton accepts $\mathit{Suff}(s)$ the set of suffixes of s. A terminal state outputs the position of its corresponding suffix. Figure 3 displays the suffix trie of $s = \mathtt{ababbb}$.

Compaction The size of a suffix trie can be quadratic in the length of s, even if pending paths are pruned (it is the case with the word $\mathtt{a}^k\mathtt{b}^k\mathtt{a}^k\mathtt{b}^k$, $k \in \mathbf{N}$). To cope with this problem, another structure is considered. It is the compacted version of the trie, called the suffix tree, and noted $ST(s)$. It keeps from the trie states that are either terminal states or forks (nodes with outdegree greater than 1). Removing other nodes leads to label arcs with words that are non-empty segments of s (see Figure 4).

It is fairly straightforward to see that the number of nodes of $ST(s)$ is no more than $2n$ (if $n > 0$), because non-terminal internal nodes have at least

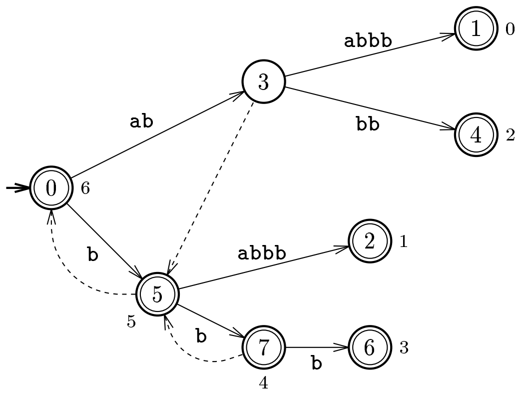
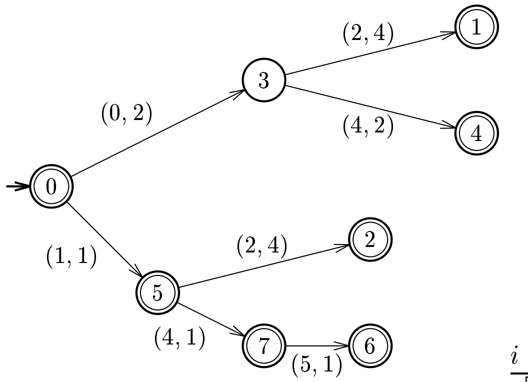


Figure 4: Suffix tree of `ababbb`.



i	0	1	2	3	4	5
$s[i]$	`a`	`b`	`a`	`b`	`b`	`b`

Figure 5: Compaction of the suffix trie of Figure 3: implementation of the suffix tree of `ababbb` of Figure 4 in which labels of arcs are represented by pairs of integers.

two children, and there are at most n external nodes. However, if the labels of arcs are stored explicitly, again the implementation can have quadratic size. The technical solution is to represent labels by pairs of integers in the form (position, length) and to keep in main memory both the tree $ST(s)$ and the textstring s (see Figure 5). The whole process yields a compacted version of the trie of suffixes that has linear size.

Minimization Another way of reducing the size of the suffix trie is to minimize it like an automaton. We then get what is called the suffix automaton $SA(s)$, which is the minimal automaton accepting $\mathit{Suff}(s)$. It is also called (suffix) DAWG's. The automaton can even be further slightly reduced by minimization if all states are made terminal, thus producing the factor automaton of the text.

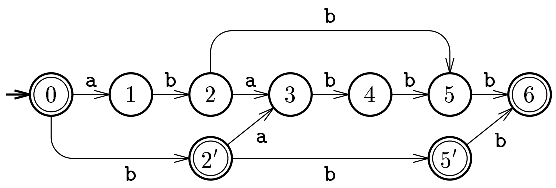


Figure 6: Suffix automaton of `ababbb`: minimal deterministic automaton accepting *Suff* (s).

Certainly the most surprising property of suffix automata, discovered by Blumer *et al.* (1983), is the linear size of the automaton. More accurately, it satisfies the inequalities:

$$|s| + 1 \leq \#states \leq 2|s| - 1,$$

$$|s| \leq \#arcs \leq 3|s| - 4.$$

Efficient constructions The construction of suffix structures can be carried on in linear time. Indeed, running times depend on the implementation of the structures, and mainly on that of the transition function. If arcs are implemented by sets of successors, transitions are done by symbol comparisons, which leads to a $O(|s| \log \operatorname{card} \Sigma)$ construction time within $O(|s|)$ memory space. This is the solution to choose for unbounded alphabets. If arcs are realized by a transition table which assumes that the alphabet is fixed, transitions are done by table lookups and the construction time becomes $O(|s|)$ using however $O(|s| \operatorname{card} \Sigma)$ memory space. These two techniques are referred to as the comparison model and the branching model respectively.

Classical algorithms that build suffix trees are by Weiner [32], McCreight [33], and Ukkonen [28]. The latter algorithm is the only one to process the text in a strictly online manner. DAWG construction was first designed by Blumer *et al.* and later extended to suffix and factor automata (see [34] and [35]).

To complete this section, we compare the complexities of the above structures to the suffix array designed by Manber and Myers [36]. A preliminary version of the same idea appears in the PAT system of Gonnet *et al.* [37]. A suffix array is an alternative implementation of the set of suffixes of a text. It consists both of a table storing the permutation of suffixes in lexicographic order, and of a table storing the maximal lengths of common prefixes between pairs of suffixes (LCP table). Access to the set of suffixes is managed via a binary search with the help of the LCP table. Storage space is obviously $O(|s|)$, access time is only $O(p + \log |s|)$ to locate a pattern of length p (it would be $O(p \times \log |s|)$ without the LCP table). Efficient preprocessing is the most difficult part of the entire implementation, it takes $O(|s| \log |s|)$ time although the total size of suffixes is $O(|s|^2)$.

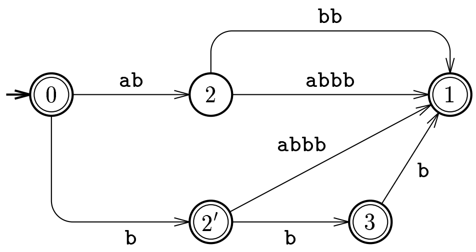
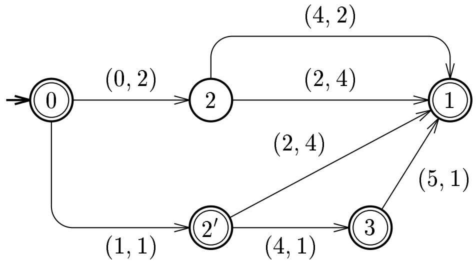


Figure 7: Compact suffix automaton of `ababbb` with explicit labels on arcs.



i	0	1	2	3	4	5
$s[i]$	a	b	a	b	b	b

Figure 8: Compact suffix automaton of `ababbb`. It is the compacted version of $SA(s)$ and the minimized version of $ST(s)$. Labels of arcs are represented by pairs of integers as in the suffix tree, see Figure 5.

Efficient storage Among the many implementations of suffix structures, we can mention the notion of sparse suffix trees due to Kärkkäinen and Ukkonen [38] which considers a reduced set of suffixes, the suffix cactus due to Kärkkäinen [39], who degenerates the suffix tree structure without increasing too much the access time, and the version dedicated to external memory (*SB-trees*) by Ferragina and Grossi [40], but several other variations exist (see [41] and [42], for example).

An excellent solution to save on the size of suffix structures is to simultaneously compact and minimize the suffix trie. Compaction and minimization are commutative operations, and when both are applied, they yield the compact suffix automaton, denoted by $CSA(s)$. Figures 7 and 8 display an example of compact suffix automaton. The direct construction of the compact suffix automaton $CSA(s)$ is possible without building first the suffix automaton $SA(s)$ nor the suffix tree (see [43]). It can be realized with the same time and space as that of other structures.

Table 1 gives an idea of the minimum and maximum sizes of suffix structures (in the comparison model). The average analysis of suffix automata, including their compact version, was done by Blumer *et al.* [44] and later completed by Raffinot [45].

The size of an implementation of the above structures is often evaluated by

Text of length n	Number of states		Number of arcs	
	min	max	min	max
Suffix trie	$n+1$	$O(n^2)$	$2n$	$O(n^2)$
Suffix Tree	$n+1$	$2n+2$	n	$2n+1$
Suffix Automaton	$n+1$	$2n-1$	n	$3n-4$
Compact SA	2	$n+1$	n	$2n-2$

Table 1: Compared sizes of suffix structures.

the average number of bytes necessary to store one letter of the original text. It is commonly admitted that these ratios are 4 for suffix arrays, 9 to 11 for suffix trees and slightly more for suffix automata, provided the text is not too large (of the order of a few megabytes).

Kurtz [46] provides several implementations of suffix trees having this performance. Holub [47] designs an implementation of compact suffix automata having ratio 5, a result that is extremely good compared to the space for a suffix array. Recently, Balík [48] gives an implementation of another type of suffix DAWG, whose ratio is only 4 and sometimes even less.

Indexing for approximate matchings Though approximate pattern matching is much more important than exact string matching for treating real sequences, it is quite surprising that no specific data structure exists for this purpose. Therefore, indexing strategies for approximate pattern matching use the data structures presented above and adapt the search procedure. This one is then based on the next result.

Lemma 1 *If x and s match with at most e differences, then x and s must have at least one identical substring of length $r = \lfloor max\{|x|, |s|\}/(e+1) \rfloor$.*

An original solution has been proposed by Manber and Baeza-Yates [49] who considered the case where the pattern embeds a string of at most e wild cards, *i.e.*, has the form $x = u\phi^i v$, where $i \le e$, $u, v \in \Sigma^*$ and $|u| \le p$ for some given e and m. Their algorithm is off-line (on the text) in the sense that the text s is preprocessed to build the suffix array associated with it. This operation costs $O(n \log |\Sigma|)$ time in the worst case. Once this is done, the problem reduces to one of efficient implementation of 2-dimensional orthogonal range queries.

Some other solutions preprocess the text to extract its q-grams or q-samples. These, possibly their neighbors up to some distance, are memorized in a straightforward data structure. This is the strategy used, for example, by the two famous programs, FastA and BLAST, which makes them run fairly fast.

There is a survey on this aspect of indexing techniques by Navarro [50].

3.4 Structural motifs

Real motifs in biological sequences are often not just simple strings. They are sometimes composed of several strings that come in organized fashion along the sequence at bounded distances from one another. Possible variations of bases can be synthetized by regular expressions. There exist efficient methods allowing to locate motifs described in this manner.

Motifs can also be repetitions of a single seed (tandem repeats) or (biological) palindromes, again with possible variations on individual bases. Palindromes for instance represent the basic elements of the secondary structures of RNA sequences. Contrary to the previous type of motifs, a regular expression cannot deal with repetitions and palindromes (at least if there is no assumption on their length).

A typical problem one may wish to address concerns the localization of tRNAsin DNA sequences. It is an instance of a wider problem which is related to the identification of functional regions in genomic sequences. The problem is to find all positions of potential tRNAsin a sequence, given a model obtained from an alignment of experimentally identified tRNAs.

There are basically two approaches to solve the question: one consists of a general-purpose method integrating searching and folding, the other consists of a self-contained method specifically designed for tRNAs. The latter produces more accurate results and faster programs. This is really needed to explore complete genomes. We briefly describe the strategy implemented by the program FAStRNA of El Mabrouk and Lisacek (see [51] for more information on other solutions), an algorithmical improvement on the tRNAscan algorithm by Fichant and Burks (1991).

FAStRNA depends on two main characteristics of tRNAs(at least of the tRNAsin the training set used by the authors): the relative invariance of some nucleotides in two highly conserved regions forming the $T\Psi C$ and D signals; the cloverleaf structure composed of four stems and three loops (see Figure 9).

In a preliminary step, the program analyzes the training set to build consensus matrices on nucleotides. This provides the invariant bases of the $T\Psi C$ and D regions used to localize the two signals. After discovering a signal, the program tries to fold the stem around it. Other foldings are performed to complete the test for the current position in the DNA sequence. Various parameters help tuning the program to increase its accuracy, and an appropriate hierarchy of searching operations enables to decrease the running time of the program.

The built-in strategy produces a very low rate of false positives and false negatives. Essentially, it fails for tRNAscontaining a very long intron. Searching for signals is implemented by a fast approximate matching procedure of the type described above, and folding corresponds to doing an alignment as presented earlier. The program runs 500 times faster than previous tRNAsearching programs.

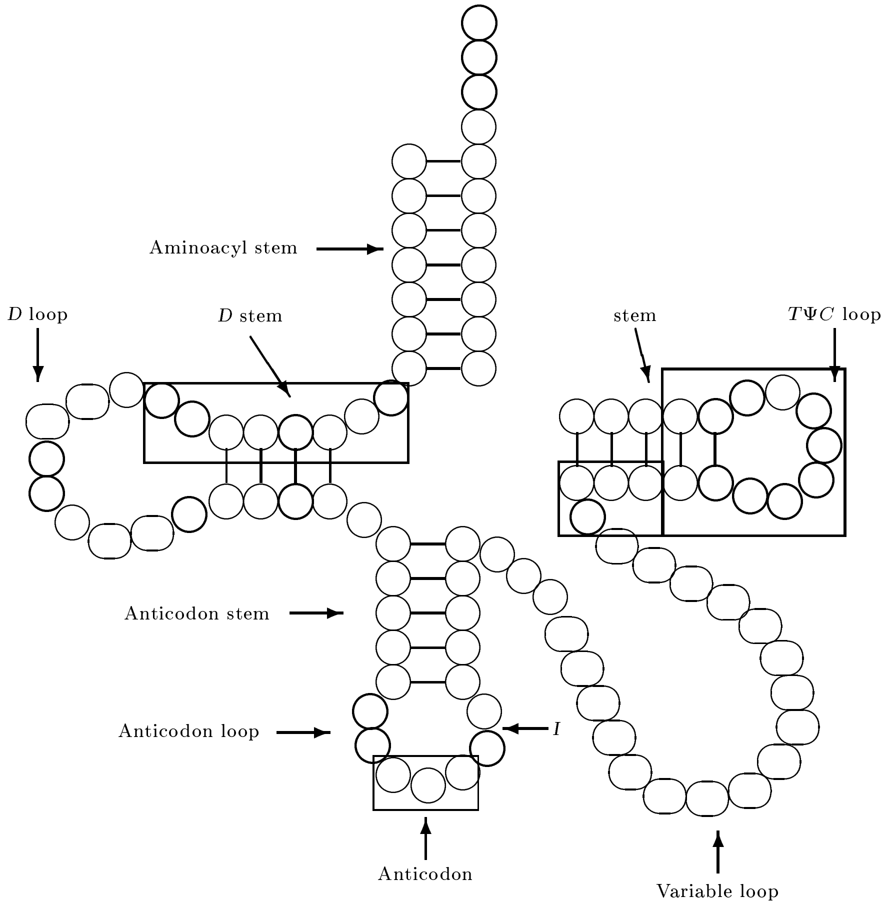


Figure 9: Cloverleaf secondary structure of a tRNA.

4 Repeated motifs identification

4.1 Exact repetitions

General algorithms

One of the first methods enabling to discover exact repetitions in strings has been designed by Karp, Miller and Rosenberg [5]. Their algorithm (henceforward called KMR) runs in $O(n \log n)$ on a string of length n but can not find all repetitions. However, various solutions based on closely related ideas have been proposed by Crochemore [52], Apostolico and Preparata [53], and Main and Lorentz [54]. They all take $O(n \log n)$ time, and any algorithm that lists all occurrences of squares, or even maximal repetitions in a string, takes at least $\Omega(n \log n)$ time because, for example, Fibonacci words contain that many occurrences of repetitions (see [52]).

A more specific question arises when one considers the problem of detecting and locating the squares (words of the form uu, for a non-empty string u) that possibly occur within a given string of length n. The lower bound for testing squarefreeness of a string is also $\Omega(n \log n)$ on general alphabets (see [54]). However, on a fixed alphabet Σ the problem of testing an occurrence of a square can be done in $O(n \log |\Sigma|)$, which implies linear-time algorithms if the size of the alphabet is fixed (see [11]). Recently, Kolpakov and Kucherov [55] proposed a linear-time algorithm to compute all the distinct segments of a string that are repetitive. A solution based on the use of a suffix tree is due to Stoye and Gusfield [56].

In the next section, we describe in some detail the KMR algorithm. Although this is not the most efficient method for finding all exact repeats, it is a very elegant algorithm and, more importantly, it allows for an easy generalization to more flexible types of repeats.

A powerful algorithm for identifying dispersed exact repeats – KMR

The original algorithm Given a string s, KMR solves the following problems.

Problem 4.1 *Identify the positions of all factors of a fixed length k that appear repeated in s.*

Problem 4.2 *Find the length k_{max} of the longest repeated factor in s, and solve problem 4.1 for $k = k_{max}$.*

KMR rests on the definition of an equivalence relation given in section 2.2. Problem 4.1 and the second part of problem 4.2 can then be formulated as the problem of finding the partition associated with E_k. Problem 4.2 further requires finding the maximum value of k such that E_k is not the identity. The algorithm is based on an iterative construction of partitions E_l for $l \leq k$. The mechanism for performing such constructions rests on the following lemma.

Lemma 4.1 *Given $a, b \geq 1$ two integers with $b \leq a$, and i, j two different positions in s such that $i, j \leq n - (a + b) + 1$, then:*

$$i \; E_{a+b} \; j \Leftrightarrow i \; E_a \; j \text{ and } (i + b) \; E_a \; (j + b).$$

The main idea behind the KMR algorithm is to use the lemma with $a = b$ for as long as possible. The lemma is consequently called the doubling lemma. This means finding repeats of length $2a$ by using previously acquired information on the repeats of length a that may become the prefixes and suffixes of those of length $2a$. If we are dealing with problem 4.1, and if k is not a power of 2, we then use the lemma with $b < a$ in a last step in order to obtain E_k. If we are treating problem 4.2, we may need more than one step to find the value of k_{max} such that $E_{k_{max}}$ is not the identity but $E_{k_{max+1}}$ is. The search for k_{max} from the smallest power of two that is bigger than k_{max}, let us say it is 2^p, can be done by applying the lemma with $b < a$ in a binary search fashion between 2^{p-1} and 2^p.

Building the partitions E_a basically corresponds to performing a set intersection operation. The intersections may be implemented using, for instance, stacks. More precisely, we need an array V_a of size n which stores, for each position i in s, the label of the class of E_a to which the a-long factor starting at i belongs. The lemma is applied by means of two arrays of stacks P and Q. Stacks in P are filled by traversing V_a. Such stacks are in fact a dual of V_a. Each one corresponds to a class c of E_a and contains the positions i in s belonging to c. Array P serves therefore to sort the prefixes of length a of the repeats of length $2a$ one is trying to identify. The content of each stack of P in turn is then poured into the appropriate stack of Q. A division separates, within a same stack of Q, elements coming from different stacks of P. Like P, array Q has as many stacks as there are classes in E_a. It serves to sort the suffixes of length also a of the repeats of length $2a$. One then just needs to orderly pour Q into V_{2a} the obtain the classes of E_{2a} checking the quorum as one goes.

As mentioned, KMR time complexity is $O(n \log k)$. When solving problem 4.2, this leads to an $O(n \log n)$ complexity because of possible degenerate cases (such as that of a string s composed of a single letter). KMR space complexity is $O(n)$.

Non-transitive relations without errors KMR may be adapted to deal with a non transitive relation R [6]. The problems solved are the same as for KMR.

Lemma 4.1 applies analogously, except that one just needs to substitute relation E by R.

Lemma 4.2 *Given $a, b \geq 1$ two integers with $b \leq a$, and i, j two different positions in s such that $i, j \leq n - (a + b) + 1$, then*

$$i \; R_{a+b} \; j \Leftrightarrow i \; R_a \; j \text{ and } (i + b) \; R_a \; (j + b).$$

Computing relations R_l for $l \leq k$ requires the same structures as for KMR, except that, as we saw, a set of positions pairwise-related by R_l is no longer an equivalence class but a clique. The algorithm was in consequence called KMRC (the "C" standing for Clique) [6]. In particular, a position may belong to two or more distinct cliques of R_l. Array V_l must now therefore be an array of stacks, like P and Q. It indicates, for each cell i corresponding to a position in s, the cliques of relation R_l to which i belongs.

The construction itself follows the same schema as indicated for KMR. Some of the sets of similar factors obtained at the end of each step may not be maximal. A further operation is therefore needed to eliminate sets included in another one so as to get maximal cliques at the end.

To calculate the complexity of the KMRC algorithm, we need to define a parameter g that measures the "degree of non-transitiveness" of relation R.

Definition 4.1 *Given R, a non-transitive relation on Σ, we call g the greatest number of cliques of R to which a symbol may belong, that is:*

$$g = Max\ \{g_a \mid a \in \Sigma,\ g_a = \textit{number of cliques to which a belongs}\}.$$

We call $\bar{g}$ the average value of g_a for $a \in \Sigma$, that is:

$$\bar{g} = \frac{\sum_a g_a}{n_c},$$

where n_c is the number of cliques of R.

If one does not count the set inclusion operations to eliminate non-maximal cliques, KMRC has time complexity $O(ng^k \log k)$ since each position i in s may belong to at most g^k (or, on the average, $\bar{g}^k$) cliques of R_k. Inclusion tests based on comparing the positions contained in each set take $O(n^2 g^{2k})$ time at the end of step k. At least another approach for testing set inclusion is possible and may result in a better theoretical (but not necessarily better in practice – this is discussed in [6]) time complexity. Space complexity is $O(ng^k)$.

4.2 Inexact repetitions – The particular case of tandem arrays (satellites)

Model for tandem arrays (satellites)

Tandem arrays (called *tandem repeats* when there are only two units) are a sequence of repeats that appear adjacent in a string. As concerns biology, such tandemly repeated units are divided into three categories depending on the length of the repeated element, the span of the repeat region and its location within the chromosome [57]. Repeats occurring in or near the centromeres and telomeres are called simply *satellites*. Their span is large, up to a million bases, and the length of the repeated element varies greatly, anywhere from 5 to a few hundreds of base pairs. In the remaining, euchromatic region, of the chromosome the kinds of tandem repeats found are classified as either *micro* or

```
GTTGCTAGAGGAAGATGGGGTTGGTACTGGTGCTACAGAGCTTTC
AGTGGTGGAGCTGGAT         ACTGGAGCAGAAGAGCTTTC
AGTAGTAGAGCTTGATGGGGTTGGTACTGGAACAGAAGAGCTTTC
AGTGCTAGAGCTGAATGGGGTTGAAGATGGAGCGGAGGAAGTGAT
GTTGCTAGAGGAAGATGGGGTTGGTACTGGTGCTACAGAGCTTTC
AGTAGTAGAGCTTGATGGGGTTGGTACTGGAGCAGAAGAGCTTTC
GGTAGTAGAGCTGGATGGAGTTGGCACTGGAGCAGAAGAGCTTTC
AGTAGTAGAGCTGGATGGAGTTGGTACTGGAGCAGAAGAGCTTTC
AGTGGTAGAGCTGG      TT   ACTGGAGCAGAAGAGCTTTC
AGTAGTAGAGCTGGATGGAGTTGGTACTGGAGCAGAAGAGCTTTC
AGTAGTAGAGCTGGATGGAGTTGGTACTGGAGCAGAAGAGCTTTC
AGTAGTAGAGCTGGATGGAGTTGGTACTGGAGCAGAAGAGCTTTC
AGTGGTAGAGCTGG      TT   ACTGGAGCAGAAGAGCTTTC
AGTGGTAGAGCTGG      TT   ACTGGAGCAGAAGAGCTTTC
AGTAGTAGAGCTGGATGGAGTTGGTACTGGAGCAGAAGAGCTTTC
AGTAGTAGAGCTGGATGGGGTTGGTACTGGAGCAGAAGAGCTTTC
AGTAGTAGAGCTGGATGGAGTTGGTACTGGAGCAGAAGAGCTTTC
AGTGGTAGAGCTGG      TT   ACTGGAGCAGAAGAGCTTTC
AGTAGTAGAGCTTGATGGGGTTGGAGCTGGAGCAGAAGAGCTTTC
AGTAGTAGAGCTTGATGGAGTTGGTACTGGAGCAGAAGAGCTTTC
AGTAGTAGAGCTTGATGGAGTTGGCACTGGAGCAGAAGAGCTTTC
AGTGGTGGAGCTGG      TT   ACTGGAGTAGAAGAGCTTTC
AGTAGTAGAGCTGGATGGAGTTGGTACTGGAGCAGAAGAGCTTTC
AGTGGTAGAGCTGG      TT   ACTGGAGCAGAAGAGCTTTC
AGTGGTGGAGCTTGATGGGGTTGGAGCTGGAGCAGAAGAGCTTTC
AGTAGTAGAGCTGGATGGAGTTGGTACTGGAGCAGAAGAGCTTTC
AGTGGTAGAGCTGG      TT   ACTGGAGCAGAAGAGCTTTC
AGTAGTAGAGCTGGATGGAGTTGGTACTGGAGCAGAAGAGCTTTC
AGTGGTAGAGCTGG      TT   ACTGGAGCAGAAGAGCTTTC
AGTAGTAGAGCTGGATGGAGTTGGTACTGGAGCAGAAGAGCTTTC
AGTGGTAGAGCTGG      TT   ACTGGAGCAGAAGAGCTTTC
AGTGGTAGAGCTGG      TT   ACTGGAGCAGAAGAGCTTTC
AGTGGTAGAGCTGG      TT   ACTGGAGCAGAAGAGCTTTC
AGTGGTAGAGCTGG      TT   ACTGGAGCAGAAGAGCTTTC
AGTAGTAGAGCTGGATGGAGTTGGTACTGGAGCAGAAGAGCTTTC
AGTGGTAGAGCTGG      TT   ACTGGAGCAGAAGAGCTTTC
AGTGGTAGAGCTGG      TT   ACTGGAGCAGAAGAGCTTTC
AGTAGTAGAGCTTGATGGGGTTGGTACTGGAGCAGAAGAGCTTTC
AGTGGTAGAGCTGG      TT   ACTGGAGCAGAAGAGCTTTC
AGTAGTAGAGCTGGATGGAGTTGGTACTGGAGCAGAAGAACTTTC
AGTAGTAGAGCTTGATGGGGTTGGTACTGGAGTAGTAGTCTTCTT
```

Figure 10: An example of a tandem repeat in chromosome IX of yeast *Saccharomyces cerevisiae*, starting at position 391 131.

mini satellites, according to the length of the repeated element. Micro satellites are composed of short units, of 2 to 5 base pairs, in copy numbers in general around 100. Mini satellites on the other hand involve slightly longer repeats, typically around 15 base pairs, in clusters of variable sizes, comprising between 30 and 2000 elements.

Figure 10 shows an example of a tandem repeat starting at position 391131 on chromosome IX from yeast (in the sequence as recovered from the ftp site ftp://ftp.mips.embnet.org/pub/yeast/). This repeat is composed of 41 full units, 16 of which present a deletion of 9 bases against the other elements. Apart from this, the repeat is well conserved overall (on the average, one mutated base per element), except for the first six units and for the last one. The repeat is located inside a coding region (in the other strand) corresponding to a glucoamylase s1/s2 precursor protein (SwissProt id: AMYH_YEAST).

Satellites of whatever type ask for a more complex definition of models than that given in Section 2.4, requiring additional constraints.

We have in fact two definitions related to a satellite model, one called *prefix model* and the other *consensus model*. This latter concerns satellite models strictly speaking while prefix models are in fact models for approximately periodic repetitions that are not necessarily tandem.

Formally, a *prefix model* of a satellite is a string $m \in \Sigma^*$ (or $\mathcal{P}(\Sigma)$) that approximately matches a train of wagons. A *wagon* of m is a factor u in s such that $dist(m, u) \leq e$. A *train* of a satellite model m is a collection of wagons $u_1, u_2, \ldots, u_p$ ordered by their starting positions in s and satisfying the following properties.

Property 1 *$p \geq min\_repeat$, where min_repeat is a fixed parameter that indicates the minimum number of elements a repeating region must contain.*

Property 2 *$left_{u_{i+1}} - left_{u_i} \in JUMP$, where $left_u$ is the position of the left-end of wagon u in s and*

$$JUMP = \{y : y \in \cup_{x\in[1,max\_jump]}\, x \times [min\_range, max\_range]\},$$

with the three parameters min_range, max_range and max_jump fixed.

A prefix model m is said to be *valid* if there is at least one train of m in the string s. Similarly, a train, when viewed simply as a sequence of substrings of s, is valid if it is the train for some model m. A prefix model represents the invariant that must be true as we progressively search for our final goal, which is to arrive at a *consensus model*. This is a prefix model which further satisfies the following property.

Property 3 *$left_{u_{i+1}} - right_{u_i} \in GAP$, where $right_u$ is the position of the right-end of wagon u, and*

$$GAP = \{y : y \in \cup_{x\in[0,max\_jump-1]}\, x \times [min\_range, max\_range]\}.$$

Parameter max_jump allows us to deal with very badly conserved elements inside a satellite (by actually not counting them) while we require that the satellite be relatively well conserved globally. Fixing max_jump at a value strictly greater than one, means that we allow some wagons (the badly conserved ones) to be "jumped over". This may be seen as "meta-errors", that is as errors involving not a single letter inside a wagon but a wagon inside a train. Note that $0 \in GAP$. This guarantees that, when jumps are not authorized, the repeats found are effectively tandem.

Since mutations affecting a unit concern *indels* (that is, insertions and deletions) as well as substitutions, it is sometimes interesting to work with a variant of the above properties where $JUMP$ and GAP are defined as

$$JUMP = \left\{y : \begin{array}{l} y \in [min\_range, max\_range] \text{ or} \\ y \in \cup_{x\in[2,max\_jump]}\, x \times [min\_range - g, max\_range + g] \end{array} \right\}$$

$$GAP = \{y : \begin{array}{l} y \in [min\_range, max\_range] \; or \\ y \in \cup_{x\in[1,max\_jump]} \; x \times [min\_range - g, max\_range + g] \end{array} \},$$

and $g \geq e$ is a fixed value. The idea is to allow the length of the badly conserved elements to vary in a larger interval than permitted for the detection of "good" wagons.

The satellite problem we propose to solve is the following.

Problem 1 *Given a string s and parameters min_repeat, min_range, max_range, max_jump, and e (possibly also g), find all consensus models m that are valid for s, and for each such m.*

In fact, the original papers [16] [58] report a set of disjoint "fittest" trains realizing each model, given a measure of "fitness".

The algorithm presented below is the only combinatorial, non-heuristical developed so far for identifying tandem arrays. Other exact approaches either treat the case of tandem repeats only [13] [14], do not allow for errors [59] [52] [60] [43], or require generating all possible (not just valid) models of a given length [61] [62] [63].

Building prefix satellite models

As with all previous cases considered in this paper, satellite models are constructed by increasing lengths. In order to determine if a model is valid, we must have some representation of the train or wagons that make it so. There are two possibilities:

- we can keep track of each valid train and its associated wagons, or
- we can keep track of individual wagons, and, on the fly, determine if they can be combined into valid trains.

The first possibility is appealing because model extension is straightforward. We would just have to verify, for each wagon of each train, whether it can be extended according to the extended model, and then count how many wagons remain to check whether the train it belonged to is still a valid train. However, there are generally many overlapping trains involving many of the same wagons for a given model. Common wagons may be present more than once in the list of occurrences of m if this is kept as a list of trains. This approach entails redundancies that lead to an inefficient algorithm. We therefore adopt the second approach, of keeping track of wagons and determining if they can be assembled into trains as needed.

The rules of prefix-model extension are given in Lemma 2 below. A wagon is identified by a triple (i, j, d) indicating that it is the substring $s_i s_{i+1} \ldots s_j$ of s and that it is $d \leq e$ differences away from its model. Position i indicates the left-end of the wagon, and j its right-end. Contrary to the other algorithms presented in this paper, models and their occurrences (the wagons) will be extended to the left. This is just to facilitate verifying Property 2. Strictly speaking, we should then speak of suffix-models instead of prefix ones. Right ends of occurrences are calculated but are used only for checking Property 3.

Lemma 2 *The triple (i, j, d) encodes a wagon of $m' = \alpha m$ with $\alpha \in \Sigma$ and $m \in \Sigma^k$ if and only if at least one of the following conditions is true:*

(match)	$(i+1, j, d)$ *is a wagon of* m *and* $s_i = \alpha$;
(substitution)	$(i+1, j, d-1)$ *is a wagon of* m *and* $s_i \neq \alpha$;
(deletion)	$(i, j, d-1)$ *is a wagon of* m;
(insertion)	$(i+1, j, d-1)$ *is a wagon of* αm;

and, furthermore, $d \leq e$.

For each prefix-model m, we keep a list of wagons of m that are in at least one train validating m. We describe such wagons as being valid with respect to m. When we extend a model (to the left) to $m' = \alpha m$, we perform two tasks:

- First, determine which valid wagons of m can be extended as above to become wagons of m'.
- Second, of these newly determined wagons of m', we keep only those that are valid with respect to m'. This requires effectively assembling wagons into trains, something that is not needed in an approach that would keep track of trains directly.

Note that we need not actually enumerate the trains in the second step, we simply must determine if a wagon is part of one. This will allow us to perform an extension step in time linear with respect to the string length.

As a final insight, consider the directed graph $G = (V, E)$ where V is the set of all valid wagons and there is an edge from wagon u to v if $\mathit{left}_v - \mathit{left}_u \in \mathit{JUMP}$. Then a wagon u is valid if it is part of a path of length $\mathit{min\_repeat}$ or more in G. Determining this property is quite simple as the graph is clearly acyclic. In the computation that follows, we effectively compute both the length of the longest path to u in Lcnt_u and the length of the longest path from u in Rcnt_u. If $\mathit{Lcnt}_u + \mathit{Rcnt}_u > \mathit{min\_repeat}$ then u is valid.

Consensus satellite models

We encode the collection of all wagons of m in a set, $L_m \subseteq \{1 \ldots, n\}$, and an $(n+1) \times (2e+1)$-element array D_m as follows:

1. $i \in L_m$ if and only if i is the left-end of at least one wagon valid with respect to m,
2. for each $i \in L_m$, the value $D_m[i, \delta]$ for $\delta \in [-e, e]$ is the edit distance of m from wagon $s_i s_{i+1} \ldots s_{i+|m|-1+\delta}$.

Intuitively, L_m gives the left-ends of all valid wagons, which is all we need to verify Properties 1 and 2. D_m gives us the distances we need for extending models, together with the right-ends needed for verifying Property 3. Formally, $(i, i+|m|-1+\delta, d)$ is a valid wagon of m if and only if $i \in L_m$ and $d = D_m[i, \delta] \leq e$.

The complete algorithm is given below. When Extend(αm) is called, it is assumed that L_m is known along with the relevant D_m values. The routine computes these items for the extension αm and recursively for the extensions thereof. Lines 1-6 compute the set of left-ends of wagons for αm derivable from wagons of m that are valid. While Lemma 2 gives us a way to do so, recall that we are using dynamic programming to compute all extensions simultaneously. This corresponds to adding the last row to the dynamic programming matrix of s versus αm. At start, L_m gives all the positions in row $|m|$ that have value e or less (and are valid) and D_m gives their values. From these, we compute the positions in row $|m|+1$ in the obvious sparse fashion to arrive at the values $L_{\alpha m}$ and $D_{\alpha m}$.

procedure Extend(αm)

1. $L_{\alpha m} \leftarrow \emptyset$
2. **for** $i+1 \in L_m$ (in decreasing order) **do**
3.   **for** $\delta \in [-e, e]$ **do**
4. $D_{\alpha m}[i,\delta] \leftarrow \min \left\{ \begin{array}{l} D_m[i+1,\delta] + (\textbf{if } s_i = \alpha \textbf{ then } 0 \textbf{ else } 1), \\ \textbf{if } i \in L_m \textbf{ then } D_m[i,\delta+1]+1, \\ \textbf{if } i+1 \in L_{\alpha m} \textbf{ then } D_{\alpha m}[i+1,\delta-1]+1 \end{array} \right\}$
5.   **if** $min_\delta\{D_{\alpha m}[i,\delta]\} \leq e$ **then**
6. $L_{\alpha m} \leftarrow L_{\alpha m} \cup \{i\}$

7. **for** $i \in L_{\alpha m}$ (in decreasing order) **do**
8. $Rcnt[i] \leftarrow \max_{k \in (i+JUMP) \cap L_{\alpha m}}\{Rcnt[k]\} + 1$
9. **for** $i \in L_{\alpha m}$ (in increasing order) **do**
10. $Lcnt[i] \leftarrow \max_{k \in (i-JUMP) \cap L_{\alpha m}}\{Lcnt[k]\} + 1$
11. **for** $i \in L_{\alpha m}$ **do**
12.   **if** $Lcnt[i] + Rcnt[i] \leq min\_repeat$ **then** $L_{\alpha m} \leftarrow L_{\alpha m} - \{i\}$

13. **if** $L_{\alpha m} \neq \emptyset$ **then**
14.   **if** $|\alpha m| \in [min\_range, max\_range]$ **then**
15.   Record(αm)
16.   **if** $|\alpha m| < max\_range$ **then**
17.   **for** $\beta \in \Sigma$ **do**
18.   Extend($\beta\alpha m$)

Once wagons have been extended whenever possible, we have to eliminate those that are no longer valid. This is performed by Lines 7 to 12. We compute, for each position $i \in L_{\alpha m}$, the maximum number of wagons in a train starting with a wagon whose left-end is at i in $Rcnt[i]$ (including itself), and the maximum number of wagons in a train ending with a wagon whose left-end is at i in $Lcnt[i]$. The necessary recurrences are given in Lines 8 and 10 of the algorithm where we recall that $JUMP = \{y : y \in \bigcup_{x\in[1,max\_jump]} x \times [min\_range, max\_range]\}$ and $i + JUMP$ denotes adding i to each element of $JUMP$. Observe that $Rcnt[i] + Lcnt[i] - 1$ is the length of the longest train containing a wagon whose left-end is at position i.

Clearly Lines 7-10 take $O(|L_{\alpha m}||JUMP|)$ time. However, when $L_{\alpha m}$ is a very large fraction of n, one can maintain an $Rcnt(Lcnt)$-prioritized queue of the positions in $(i + JUMP) \cap L_{\alpha m}$, to obtain an $O(n\ max\_jump\ \log|JUMP|)$ bound.

Finally in the remaining steps, Lines 13-18, the algorithm calls Record to record potential models and then recursively tries to extend the model if possible. Routine Record confirms that the model is a consensus model by verifying Property 3 and recording the intervals spanned by trains that are valid for the consensus model, if any.

The total time taken by the algorithm is $O(n\ (|JUMP| + e)\ max\_range\ \mathcal{N}(e, max\_range)) = O(n\ max\_range^2\ max\_jump\ \ \mathcal{N}(e, max\_range))$ as $e < max\_range$. The term $\mathcal{N}(e, max\_range)$ corresponds to the number of words in the e-neighbourhood of a word w of length max_range, that is, words that are at a Levenshtein distance at most e from w. This number is bounded over by k^e.

The space requirement is that of keeping all the information concerning at most max_range models at a time (a model m and all its prefixes). It is therefore $O(n\ max\_range\ e)$ as only $O(n\ e)$ storage is required to record the left-end positions and edit-distance at each possible right-end.

5 Motif extraction

5.1 Spelling simple models

We now present increasingly sophisticated models and algorithms for extracting models which occur in a set of strings (possibly not all). Such models correspond in general to binding sites, that is to sites in a biological molecule that will come into contact with a site in another molecule thus permitting some biological process to start (for instance, transcription or translation). We start by considering simple models.

The problem we wish to solve is the following.

Problem 2 *Given a set of N strings $\mathcal{S} = s_1, \ldots, s_N$, an integer $e \geq 0$ and a quorum $q \leq N$, find all models m such that m is valid, that is, occurs with at most e errors in at least q strings of set $\mathcal{S}$.*

The spelling of models is done using a suffix tree. The idea comes from the observation that long strings, specially when they are defined over a small alphabet, may contain many exact repetitions. One does not want to compare such repeated parts more than once with the potentially valid models. One way of doing that is using a representation of the strings that allows to put together some of the repetitions, that is, using an index of the strings such as a suffix tree.

Trees for representing all the suffixes of a set of strings $\{s_i,\ 1 \leq i \leq N$ for some $N \geq 2\}$ are called *generalized suffix trees* and are constructed in a way very similar to the construction of the suffix tree for a single string [64] [65]. We denote such generalized trees by $\mathcal{GT}$. They share all the properties of a suffix tree given in Section 3.3 with string s substituted by strings $s_1, \ldots, s_N$.

In particular, a generalized suffix tree $\mathcal{GT}$ satisfies the fact that every suffix of every string s_i in the set leads to a distinct leaf. When p strings, $p \geq 2$, have a same suffix, the generalized tree has therefore p leaves corresponding to this suffix, each associated with a different string. To achieve this property during construction, we just need to concatenate to each string s_i of the set a symbol that is not in Σ and is specific to that string.

To be able to spell valid models (*i.e.* models satisfying the quorum constraint), we need to add some information to the nodes of the suffix tree.

In the case where we are looking for repeats in a single string s, we just need to know, for each node x of $\mathcal{T}$, how many leaves are contained in the subtree rooted at x. Let us denote $leaves_x$ this number for each node x. Such information can be added to the tree by a simple traversal of it.

If we are dealing with $N \geq 2$ strings, and therefore a generalized suffix tree $\mathcal{GT}$, it is not enough anymore to know the value of $leaves_x$ for each node x in $\mathcal{GT}$ in order to be able to verify whether a model remains valid. Indeed, for each node x, we need this time to know not only the number of leaves in the subtree of $\mathcal{GT}$ having x as root, but that number for each different string the leaves refer to.

In order to do that, we must associate to each node x in $\mathcal{GT}$ an array, denoted $colours_x$, of dimension N that is defined by:

$$colours_x[i] = \begin{cases} 1 & \text{if at least one leaf in the subtree} \\ & \text{rooted at } x \text{ represents a suffix of } s_i \\ 0 & \text{otherwise} \end{cases}$$

for $1 \leq i \leq N$.

The array $colours_x$ for all x may also be obtained by a simple traversal of the tree in which each visit to a node takes $O(N)$ time. The additional space required is $O(N)$ per node.

One must observe that occurrences are now grouped into classes and “real” ones, that is, occurrences considered as individual words in the strings, are never manipulated directly. Present case occurrences of a model are thus in fact nodes of the generalized suffix tree (we denote them by the term “node-occurrences”) and are extended *in the tree* instead of in the string. Once the process of model spelling has ended, the start positions of the “real” occurrences of the valid models may be recovered by traversing the subtrees of the nodes reached so far, and by reading the labels of their leaves.

The algorithm is a development of the recurrence formula given in the lemma below where x denotes a node of the tree, $father(x)$ its father, and d the number of errors between the label of the path going from the root to x as against a model m.

Lemma 3 *(x, d) is a node-occurrence of $m' = m\alpha$ with $m \in \Sigma^k$ and $\alpha \in \Sigma$ if, and only if, one of the following two conditions is verified:*

(match)	*$(father(x), d)$ is a node-occurrence of m and the label of the from $father(x)$ to x is α;*
(substitution)	*$(father(x), d-1)$ is a node-occurrence of m and the label of the arc from $father(x)$ to x is $\beta \neq \alpha$;*
(deletion)	*$(x, d-1)$ is a node-occurrence of m;*
(insertion)	*$(father(x), d-1)$ is a node-occurrence of $m\alpha$.*

and, furthermore, $d \leq e$.

The algorithm time complexity is $O(nN^2\mathcal{N}(e,k))$.

5.2 Structured models

Introducing structured models

Although the objects defined in the previous section can be reasonable, algorithmically tractable models for single binding sites, they do not take into account the fact that such sites are often not alone (in the case of eukaryotes, they may even come in clusters) and, specially, that the relative positions of such sites, when more than one participates in a biological process, are in general not random. This is particularly true for some DNA binding sites such as those involved in the transcription of DNA into RNA (*e.g.* the so-called promoter sequences).

There is therefore a need for defining biological models as objects that take such characteristics into account. This has the motivation just mentioned but presents also interesting algorithmical aspects: exploiting such characteristics could lead to algorithms that are both more sensitive and more efficient. Models that incorporate such characteristics are called *structured models*. They are related to structured motifs of Section 3.

Formally, a *structured model* is a pair (m, d) where:

- m is a p-tuple of simple models $(m_1, \ldots, m_p)$ (representing the p parts a structured model is composed of – we shall call these parts *boxes*),
- d is a $(p-1)$-tuple $((d_{min_1}, d_{max_1}, \delta_1), \ldots, (d_{min_{p-1}}, d_{max_{p-1}}, \delta_{p-1}))$ of triplets (representing the $p-1$ intervals of distance between two successive boxes in the structured model),

with p a positive integer, $m_i \in \Sigma^+$, and d_{min_i}, d_{max_i} ($d_{max_i} \geq d_{min_i}$), δ_i non negative integers.

Given a set of N strings $s_1, \ldots, s_N$ and an integer q, $1 \leq q \leq N$, a model (m, d) is said to be *valid* if, for all i, $1 \leq i \leq (p-1)$, and for all occurrences u_i of m_i, there exist occurrences $u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_p$ of $m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_p$ such that:

- $u_1, \ldots, u_{i-1}, u_i, u_{i+1}, \ldots, u_p$ belong to the same string of the set,

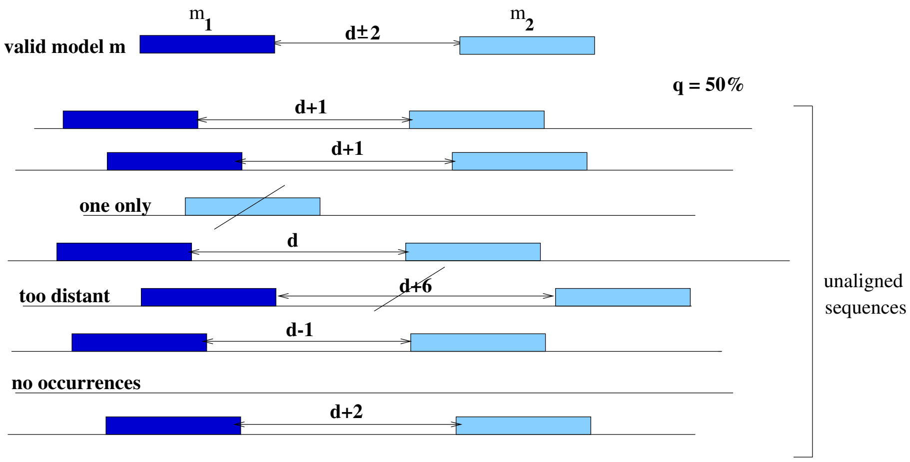


Figure 11: Example of a model with two boxes ($p = 2$).

- there exists d_i, with $d_{min_i} + \delta_i \leq d_i \leq d_{max_i} - \delta_i$, such that the distance between the end position of u_i and the start position of u_{i+1} in the string is equal to $d_i \pm \delta_i$,
- d_i is the same for p-tuples of occurrences present in at least q distinct strings.

The term d_i represents a distance and $\pm\delta_i$ an allowed interval around that distance. When $\delta_i = (d_{max_i} - d_{min_i} + 1)/2$, then δ_i is omitted, and d in a structured model (m, d) is denoted by a pair (d_{min_i}, d_{max_i}). An example of a model with $p = 2$ is given in Figure 11.

Observe that simple models are indeed but a special case of structured ones.

Statement of the structured model problem

Concerning structured models, solutions to variants of increasing generality of a same basic problem are proposed. Suffix trees are used in all cases. These variants may be stated as follows; given a set of N strings $s_1, \ldots, s_N$, a nonnegative integer e and a positive integer q.

Problem 3 *Find all models of the form* $((m_1, m_2), (d_{min_1}, d_{max_1}))$ *that are valid.*

Problem 4 *Find all models of the form* $((m_1, \ldots, m_p)$, $((d_{min_1}, d_{max_1}), \ldots,$ $(d_{min_{p-1}}, d_{max_{p-1}})))$ *that are valid, where* $p \geq 2$.

Problem 5 *Find all models of the form* $((m_1, m_2), (d_{min_1}, d_{max_1}, \delta_1))$ *that are valid.*

Problem 6 *Find all models of the form* $((m_1, \ldots, m_p), ((d_{min_1}, d_{max_1}, \delta_1), \ldots, (d_{min_{p-1}}, d_{max_{p-1}}, \delta_{p-1})))$ *that are valid, where* $p \geq 2$.

The last two problems represent situations where the exact intervals of distances separating the parts of a structured site are unknown, the only known fact being that these intervals cover a restricted range of values. How restricted is indicated by the δ_i parameters. We present below algorithms for the first two problems only. Further details on the other two may be found in [17].

To simplify matters, we shall consider that, for $1 \leq i \leq p$, $m_i \in \Sigma^k$ where k is a positive integer, *i.e.*, that each single model m_i of a structured model (m, d) is of fixed, unique length k. In a likewise manner, we shall assume that each part m_i has the same error rate e and, when dealing with models composed of more than two boxes, that the d_{min_i}, d_{max_i} and, possibly, δ_i for $1 \leq i \leq p-1$ have identical values. We denote by d_{min}, d_{max} and δ these values. Problem 4 is then formulated as finding all models $((m_1, \ldots, m_p), (d_{min}, d_{max}))$ that are valid and Problem 6 as finding all valid models $((m_1, \ldots, m_p), (d_{min}, d_{max}, \delta))$.

Besides fixing a maximum error rate for each part in a structured model, one can also establish a maximum error rate for the whole model. Such a global error rate allows to consider in a limited way possible correlations between boxes in a model.

Another possible global, or local, constraint one may wish to consider for some applications concerns the composition of boxes. One may, for instance, determine that the frequency of one or more nucleotide in a box (or among all boxes) is below or above a certain threshold. For structured models composed of more than p boxes, one may also establish that a box i is palindromic in relation to a box j for $1 \leq i < j \leq p$. In algorithmical terms, the two types of constraints just mentioned are not equivalent. The first type, box composition whether local or global, can in general be verified only *a posteriori* while the second type (palindromic boxes) will result in a, sometimes substantial, pruning of the virtual trie of models.

Introducing such additional constraints may in some cases ask for changes to the basic algorithms described below. The interested reader may find the details concerning such changes in the original papers [17] [66].

We present, in the next section, first a naive approach and then two algorithms that are efficient enough to tackle structured model extraction (Problem 3) from big datasets. The second algorithm has a better time complexity than the first but needs more space. The first is easier to understand and implement. Both are described in more detail than previous algorithms as structured models in some ways incorporate almost all other kinds of motifs we are considering. The most notable exception concerns satellites that is discussed in Section 4.2. We then show how to extend these to treat Problem 4. Details on the algorithms for solving Problems 5 and 6 may be found in [17].

Other combinatorial approaches were developed for treating somewhat similar kinds of structured motifs. They either enumerate all possible (not just valid) motifs [67], do not allow for errors [68] [69], or are heuristics [70] [71].

Algorithms for the special case of a known interval of distance

Naive approach A naive way of solving Problem 3 consists in extracting and storing all valid single models of length k (given q and e), and then, once this is finished, in verifying which pairs of such models could represent valid structured models (given an interval of distance $[d_{min}, d_{max}]$).

The lemma used for building valid single models is similar to Lemma 3 except that in practice, for most biological problems we wish to address [72] [73], substitutions only are allowed in general. The lemma therefore becomes as stated.

Lemma 4 *(x, d) is a node-occurrence of $m' = m\alpha$ with $m \in \Sigma^k$ and $\alpha \in \Sigma$ if, and only if, one of the following two conditions is satisfied:*

(match)	*$(father(x), d)$ is a node-occurrence of m and the label of the arc from $father(x)$ to x is α;*
(substitution)	*$(father(x), d-1)$ is a node-occurrence of m and the label of the arc from $father(x)$ to x is $\beta \neq \alpha$.*

and, furthermore, $d \leq e$.

One way of doing the verification profits from the simple observation that two single models m_1 and m_2 may form a structured one if, and only if, at least one occurrence of m_1 is at the right distance of at least one occurrence of m_2. Building an array of size nN where cell i contains the list of models having an occurrence starting at that position in $s = s_1 \ldots s_N$ allows to compare models in cell i to models in cells $i + d_{min}, \ldots, i + d_{max}$ only. If the sets of occurrences of models are ordered, this comparison may be done in an efficient way (in time proportional to the size of the sets of node-occurrences, which is upper-bounded by nN).

First algorithm: Jumping in the suffix tree A first non-naive approach to solving Problem 3 starts by extracting single models of length k. Since we are traversing the trie of models in depth-first fashion (also in lexicographic order), models are recursively extracted one by one. At each step, a single model m (and its prefixes) is considered. Once a valid model m_1 of length k is obtained together with its set of $\mathcal{T}$-node-occurrences V_1 (which are nodes located at level k in $\mathcal{GT}$), the extraction of all single models m_2 with which m_1 could form a structured model $((m_1, m_2), (d_{min}, d_{max}))$ starts. This is done with m_2 representing the empty word and having as node-occurrences the set V_2 given by:

$$V_2 = \{(w, e_w = e_v) \mid \exists v \in V_1 \; with \; d_{min} \leq \; level(w) - \; level(v) \leq d_{max}\},$$

where level(v) indicates the level of node v in $\mathcal{GT}$. From a node-occurrence v in V_1, a jump is therefore made in $\mathcal{GT}$ to all potential start node-occurrences w of m_2. These nodes are the d_{min}- to d_{max}-generation, descendants of v in $\mathcal{GT}$. Exactly the same recurrence formula given in Lemma 4 may be applied to the nodes w in V_2 to extract all single models m_2 that, together with m_1 could

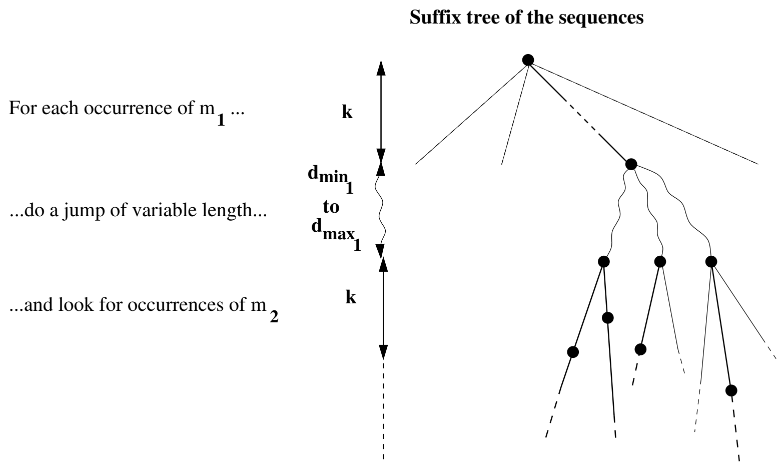


Figure 12: Extracting structured models (in the context of Problem 3) with a suffix tree – An illustration of Algorithm 1.

form a structured model verifying the conditions of the problem, for all valid m_1. An illustration is given in Figure 12 and a pseudo-code is presented below. The procedure ExtractModels is called with arguments: m equal to the empty word having as sole node-occurrence the root of $\mathcal{GT}$, and i equal to 1.

```
procedure ExtractModels(Model m, Block i)
    for each node-occurrence v of m do
        if i = 2 then
            put in PotentialStarts the children w of v at levels k + d_min to
            k + d_max
        else
            put v (i.e., the root) in PotentialStarts
    for each model m_i (and its occurrences) obtained by doing a recursive
            depth-first traversal from the root of the virtual model tree M
            while simultaneously traversing GT from the node-occurrences in
            PotentialStarts (Lemma 4 and quorum constraint) do
        if i = 1 then
            ExtractModels(m = m_1, i + 1)
        else
            report the complete model m = ((m_1, m_2), (d_min, d_max)) as valid
```

Since the minimum and maximum length of a structured model (m, d) that may be considered are, respectively, $2k + d_{min}$ and $2k + d_{max}$, we need only build the tree of suffixes of length $2k + d_{min}$ or more, and for each such suffix to consider at most the first $2k + d_{max}$ symbols.

The observation made in the previous paragraph applies also to the second algorithm (Section 5.2 below). Note that, in both cases, this implies $n_i \leq n_{i+1} \leq Nn$ for all $i \geq 1$ where n_i is the number of nodes at depth i in $\mathcal{GT}$.

Second algorithm: Modifying the suffix tree The second algorithm initially proceeds like the first: it starts by building single models of length k, one at a time. For each node-occurrence v of a first part m_1 considered in turn, a jump is made in $\mathcal{GT}$ down to the descendants of v situated at lower levels. This time however, the algorithm just passes through the nodes at these lower levels, grabs some information the nodes contain and jumps back up to level k again (in a way that is explained below). The information grabbed in passing is used to temporarily and partially modify $\mathcal{GT}$ and start, *from the root of* $\mathcal{GT}$, the extraction of the second part m_2 of a potentially valid structured model $((m_1, m_2), (d_{min}, d_{max}))$. Once the operation of extracting all possible companions m_2 for m_1 has ended, that part of $\mathcal{GT}$ that was modified is restored to its previous state. The construction of another single model m_1 of a structured model $((m_1, m_2), (d_{min}, d_{max}))$ then follows, and the whole process unwinds in a recursive way until all structured models satisfying the initial conditions are extracted.

More precisely, the operation performed between the spelling of models m_1 and m_2 locally alterates $\mathcal{GT}$ up to level k to a tree $\mathcal{GT}'$ that contains only the k-long prefixes of suffixes of $\{s_1, \ldots, s_N\}$ starting at a position between d_{min} and d_{max} from the end position in s_i of an occurrence of m_1. Tree $\mathcal{GT}'$ is, in a sense, the union of all the subtrees t of depth at most k rooted at nodes that represent start occurrences of a potential companion m_2 for m_1.

For each model m_1 obtained, before spelling all possible companions m_2 for m_1, the content of $colors_z$ for all nodes z at level k in $\mathcal{GT}$ are stored in an array L of dimension n_k (this is for later restoration of $\mathcal{GT}$). Tree $\mathcal{GT}'$ is then obtained from $\mathcal{GT}$ by considering all nodes w in $\mathcal{GT}$ that may be reached during a descent of, this time, $k + d_{min}$ to $k + d_{max}$ arcs down from the node-occurrences (v, e_v) of m_1. These correspond to all end node-occurrences (instead of start as in the first algorithm) of potentially valid models having m_1 as first part. The boolean arrays $colors_w$ for all w indicate to which input strings these occurrences belong. This is the information we grab in passing and take along the only path of suffix links in $\mathcal{GT}$ that leads back to a node z at level k in $\mathcal{GT}$. If it is the first time z is reached, $colors_z$ is assigned $colors_w$, otherwise $colors_w$ is added (boolean "or" operation) to $colors_z$. Once all nodes v and w have been treated, the information contained in the nodes z that were reached during this operation are propagated up the tree from level k to the root (using normal tree arcs) in the following way: if $\bar{z}$ and $\hat{z}$ have same parent z, then $colors_z = colors_{\bar{z}} \cup colors_{\hat{z}}$. Any arc from the root that is not visited at least once in such a traversal up the tree is not part of $\mathcal{GT}'$, nor are the subtrees rooted at its end node.

The extraction of all second parts m_2 of a structured model (m, d) follows, as for single models in the initial algorithm (Lemma 4 in Section 5.2).

Restoring the tree $\mathcal{GT}$ as it was before the operations described above requires restoring the value of $colors_z$ preserved in L for all nodes z at level k and propagating the information (state of boolean arrays) from z up to the root.

Since nodes w at level between $2k + d_{min}$ to $2k + d_{max}$ will be solicited for the same operation over and over again, which consists in following the unique suffix-link path from w to a node z at level k in $\mathcal{GT}$, $\mathcal{GT}$ is pre-treated so that one single link has to be followed from z. Going from w to z takes then constant time.

An illustration is given in Figure 13. A pseudo-code of the algorithm is as follows. The procedure ExtractModels is called, as for the first algorithm, with both arguments m equal to the empty word having as sole node-occurrence the root of $\mathcal{GT}$, and i equal to 1.

```
procedure ExtractModels(Model m, Block i)
  for each node-occurrence v of m do
     if i = 2 then
        put in PotentialEnds the children w at levels 2k + d_min to 2k+
        d_max
        for each node-occurrence w in PotentialEnds do
           follow fast suffix-link to node z at level k
           put z in L
           if first time z is reached then
              initialize colors_z with zero
              put z in NextEnds
           add colors_w to colors_z
        do a depth-first traversal of GT to update the boolean arrays from
           the root to all z in NextEnds (let GT'
           be the k-deep tree obtained by such an operation)
        if i = 1 then
           Tree = GT
        else
           Tree = GT'
  for each model m_i (and its occurrences) obtained by doing a recursive
        depth-first traversal from the root of the virtual model tree M
        while simultaneously traversing Tree from the root (Lemma 4 and
        quorum constraint) do
     if i = 1 then
        ExtractModels(m = m_1, i + 1)
     else
        report the complete model m = ((m_1, m_2), (d_min, d_max)) as a valid
        one
        restore tree GT to its original state using L
```

Proposition 1 *The following two statements are true:*

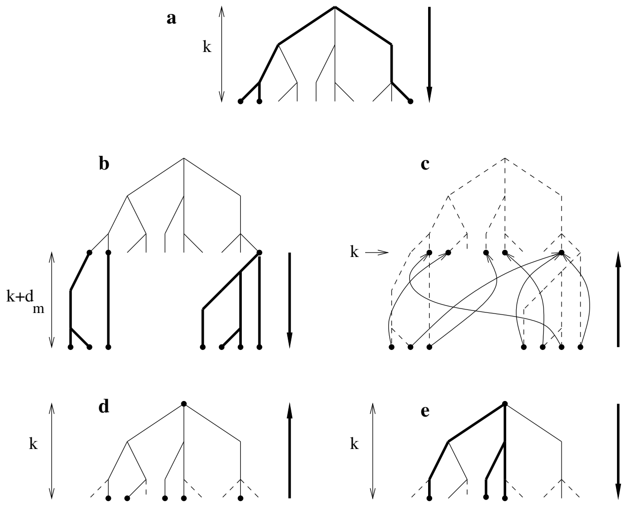


Figure 13: Extracting structured models (in the context of Problem 3) with a suffix tree – An illustration of Algorithm 2. Fig. 13a corresponds to the extraction of the first single models m_1 of structure models (m, d); Fig. 13b to the jump of $k+d_{min}$ to $k+d_{max}$ down normal tree arcs to grab some information (to lighten the figure, we made here $d_{min} = d_{max} = d_m$); Fig. 13c shows the jump back up to level k following suffix links with the information grabbed in passing; Fig. 13d represents the propagation of the information received at level k up to the root; finally Fig. 13e illustrates the search for second single models m_2 of structure models (m, d) in tree $\mathcal{T}'$.

- *$\mathcal{GT}'$ contains only the k-long prefixes of suffixes of $\{s_1, \ldots, s_N\}$ that start at a position between d_{min} and d_{max} of the end position in $\{s_1, \ldots, s_N\}$ of an occurrence of m_1;*
- *the above algorithm solves Problem 3.*

The proof is straightforward and may be found in the original papers [17] [66].

Complexity The naive approach to solving Problem 3 requires $nN^2\mathcal{N}(e,k)$ time to find single models that could correspond to either part of a structured model (and $nN\mathcal{N}(e,k)$ space to store all potential parts). If we denote by Δ the value $d_{max} - d_{min} + 1$, finding which pair of single models may be put together to produce a structured model could then be done in time proportional to:

$$\underbrace{\mathcal{N}(e,k)}_{(1)}\underbrace{\Delta\mathcal{N}(e,k)}_{(2)}\underbrace{nN}_{(3)}\underbrace{nN}_{(4)}$$

where (1) is the maximum number of single models to which a position may belong, (2) is the maximum number of models to which a position at a distance between $k + d_{min}$ and $k + d_{max}$ from the first may belong, (3) is the maximum number of comparisons that must be done to check whether two single models may form a structured one and, finally, (4) is the number of starting positions to consider.

The total time complexity of the second algorithm is $O(Nn_k\mathcal{N}^2(e,k) + Nn_{2k+d_{max}}\mathcal{N}(e,k))$. Space complexity is slightly higher than for the first algorithm: $O(N^2n + Nn_k)$ where $n_k \leq Nn$. The second term is for array L.

In either case, the complexity obtained is better both in terms of time and space than the one given by a naive solution to Problem 3.

Extending the algorithms to extract structured models having $p > 2$ parts

First algorithm: Jumping in the suffix tree Extending the first algorithm to extract structured models composed of $p > 2$ parts, that is solving Problem 4, is immediate. After extracting the first i parts of a structured model $((m_1, \ldots, m_p), (d_{min}, d_{max}))$ for $1 \leq i < p - 1$, one jumps down in the tree $\mathcal{GT}$ (following normal tree arcs) to get to the d_{min}- to d_{max}-descendants of every node-occurrence of $((m_1, \ldots, m_i), (d_{min}, d_{max}))$ then continues the extraction from there using Lemma 4.

A pseudo-code is given below.

procedure ExtractModels(Model m, Block i)
1. **for** each node-occurrence v of m **do**
2. **if** $i > 1$ **then**
3. put in $PotentialStarts$ the children w of v at levels $(i-1)k+$ $(i-1)d_{min}$ to $(i-1)k + (i-1)d_{max}$
4. **else**
5. put v (the root) in $PotentialStarts$
6. **for** each model m_i (and its occurrences) obtained by doing a recursive depth-first traversal from the root of the virtual model tree $\mathcal{M}$ while simultaneously traversing $\mathcal{GT}$ from the node-occurrences in $PotentialStarts$ (Lemma 4 and quorum constraint) **do**
7. **if** $i < p$ **then**
8. ExtractModels($m = m_1 \cdots m_i$, $i + 1$)
9. **else**
10. report the complete model $m = ((m_1, \cdots, m_p), (d_{min}, d_{max}))$ as a valid one

Second Algorithm: Modifying the Suffix Tree Extending the second algorithm to solve Problem 4 is slightly more complex and thus calls for a few remarks. The operations done to modify the tree between building m_i and

m_{i+1}, $i \geq 1$, are almost the same as those described in Section 5.2 except for two facts. One is that up to $(p-1)$ arrays L are now needed to restore the tree after each modification it undergoes. The second difference, more important, is that we need to keep, for each node v_k at level k reached from an ascent up $\mathcal{GT}$'s suffix links, a list, noted $Lptr_{v_k}$, of pointers to those nodes, at lower levels, that affected the content of v_k. The reason for this is that tree $\mathcal{GT}$ is modified up to level k only (resulting in tree $\mathcal{GT}'$) as these are the only levels concerned by the search for occurrences of each box of a structured model. Lower levels of $\mathcal{GT}$ remain unchanged, in particular the boolean arrays at each node below level k. To obtain the correct information concerning the potential end node-occurrences of boxes i for $i > 2$ (*i.e.* to which strings such occurrences belong), we therefore cannot move down $\mathcal{GT}$ from the ends of node-occurrences in $\mathcal{GT}'$ of box $(i-1)$. If we did, we would not miss any occurrence but we could get more occurrences, *e.g.* the ones that did not have an occurrence of a previous box in the model. We might thus overcount some strings and consider as valid a model that, in fact, no longer satisfied the quorum. We have to go down $\mathcal{GT}$ from the ends of node-occurrences *in* $\mathcal{GT}$, that is from the original ends of node-occurrences in $\mathcal{GT}$ of the boxes built so far. These are reached from the list of pointers $Lptr_{v_k}$ for the nodes v_k that are identified as occurrences of the box just treated. For models composed of p boxes, we need at most $(p-1)$ lists $Lptr_{v_k}$ for each node v_k at level k.

A pseudo-code for the algorithm is as follows.

procedure ExtractModels(Model m, Block i)
for each node-occurrence v of m **do**
    if $i > 2$ **then**
        put in $PotentialEnds$ the children w at levels $ik + (i-1)d_{min}$ to $ik + (i-1)d_{max}$
        for each node-occurrence w in $PotentialEnds$ **do**
            follow fast suffix-link to node z at level k
            put z in $L(i)$
            if first time z is reached **then**
                initialize $colors_z$ with zero
                put z in $NextEnds$
            add $colors_w$ to $colors_z$
        do a depth-first traversal of $\mathcal{GT}$ to update the boolean arrays from the root to all z in $NextEnds$ (let $\mathcal{GT}'$ be the k-deep tree obtained by such an operation)
    if $i = 1$ **then**
        Tree $= \mathcal{GT}$
    else
        Tree $= \mathcal{GT}'$
for each model m_i (and its occurrences) obtained by doing a recursive depth-first traversal from the root of the virtual model tree $\mathcal{M}$ while simultaneously traversing *Tree* from the root (Lemma 4 and

quorum constraint) **do**

if $i < p$ **then**

ExtractModels($m = m_1 \cdots m_i$, $i+1$)

else

report the complete model $m = ((m_1, \cdots, m_p), (d_{min}, d_{max}))$ as a valid one

if $i > 1$ **then**

restore tree $\mathcal{GT}$ to its original state using $L(i)$

Complexity The first algorithm requires $O(Nn_{pk+(p-1)d_{max}}\ \mathcal{N}^p(e,k))$ time, where $\mathcal{N}^p(e,k)) \leq k^{pe}|\Sigma|^{pe}$. The space complexity remains the same as for solving Problem 1, that is $O(N^2n)$.

The total time complexity of the second algorithm is $O(Nn_k\mathcal{N}^p(e,k) + Nn_{pk+(p-1)d_{max}}\ \mathcal{N}^{p-1}(e,k))$. The space complexity is $O(N^2n + N(p-1)n_k)$.

Acknowledgements

We are grateful to N. El Mabrouk for helpful discussions, and to our research partners: C. Allauzen, A. Vanet, L. Marsan, M. Raffinot, A. Viari.

References

[1] R. Durbin, S. R. Eddy, A. Krogh and G. Mitchison. *Biological Sequence Analysis. Probabilistic Models of Proteins and Nucleic Acids*, Cambridge University Press, 1998.

[2] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[3] J.C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*, PWS Publisher, 1996.

[4] M.S. Waterman. *Introduction to Computational Biology. Maps, Sequences and Genomes.* Chapman and Hall, 1995.

[5] R.M. Karp, R.E. Miller, and A.L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. pages 125–136. Proc. 4th Annu. ACM Symp. Theory of Computing, 1972.

[6] H. Soldano, A. Viari, and M. Champesme. Searching for flexible repeated patterns using a non transitive similarity relation. *Pattern Recognition Letters*, 16:233–246, 1995.

[7] M.-F. Sagot, V. Escalier, A. Viari, and H. Soldano. Searching for repeated words in a text allowing for mismatches and gaps. In R. Baeza-Yates and U. Manber, editors, *Second South American Workshop on String Processing*, pages 87–100, Viñas del Mar, Chili, 1995. University of Chili.

[8] G. Salton. *Automatic text processing.* Addison-Wesley, 1989.

[9] R. Baeza-Yates and B. Ribero-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

[10] A. Apostolico and Z. Galil, Eds. Pattern Matching Algorithms. Oxford University Press, New York (1997).

[11] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, 1994.

[12] G.A. Stephen, *String Searching Algorithms*, World Scientific, 1994.

[13] G. Landau and J. Schmidt. An algorithm for approximate tandem repeats. In Z. Galil A. Apostolico, M. Crochemore and U. Manber, editors, *Combinatorial Pattern Matching*, volume 684 of *Lecture Notes in Computer Science*, pages 120–133. Springer-Verlag, 1993.

[14] S. K. Kannan and E. W. Myers. An algorithm for locating non-overlapping regions of maximum alignment score. In Z. Galil A. Apostolico, M. Crochemore and U. Manber, editors, *Combinatorial Pattern Matching*, volume 684 of *Lecture Notes in Computer Science*, page 7486. Springer-Verlag, 1993.

[15] S. Kurtz, E. Ohlebusch, C. Schleiermacher, J. Stoye and R. Giegerich. Computation and visualization of degenerate repeats in complete genomes, In *Eight International Symposium on Intelligent Systems for Molecular Biology*, La Jolla, California, 2000. AAAI Press.

[16] M.-F. Sagot and E. W. Myers. Identifying satellites and periodic repetitions in biological sequences. *J. of Computational Biology*, 10:10–20, 1998.

[17] L. Marsan and M.-F. Sagot. Algorithms for extracting structured motifs using a suffix tree with an application to promoter and regulatory site consensus identification. *J. Computational Biology*, 7:345-362, 2000.

[18] M.-F. Sagot, A. Viari, and H. Soldano. A distance-based block searching algorithm. In C. Rawlings, D. Clark, R. Altman, L. Hunter, T. Lengauer, and S. Wodak, editors, *Third International Symposium on Intelligent Systems for Molecular Biology*, pages 322–331, Cambridge, England, 1995. AAAI Press.

[19] M.O. Dayhoff, R.M. Schwartz, and B.C. Orcutt. A model of evolutionary change in proteins. In M.O. Dayhoff, editor, *Atlas of Protein Sequence an Structure*, volume 5 suppl.3, pages 345–352. Natl. Biomed. Res. Found., 1978.

[20] S. Henikoff and J.G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc. Natl. Acad. Sci. USA*, 89:10915–10919, 1992.

[21] J.L. Risler, M.O. Delorme, H. Delacroix, and A. Hénaut. Amino acid substitutions in srtucturally related proteins: a pattern recognition approach. *J. Mol. Biol.*, 204:1019–1029, 1988.

[22] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[23] M. J. Fischer and M. Paterson. String matching and other products. SIAM-AMS Complexity of Computation, R. M. Karp, ed., Providence, RI, 1974, 113–125.

[24] A. Apostolico and R. Giancarlo. Sequence alignment in molecular biology. *Journal of Computational Biology* **5**:2 (1998) 173–196.

[25] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[26] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.

[27] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. A basic local alignment search tool. *J. Mol. Biol.*, 215:403–410, 1990.

[28] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[29] R.A. Baeza-Yates and C. Perleberg. Fast and practical aproximate string matching. In Z. Galil A. Apostolico, M. Crochemore and U. Manber, editors, *Combinatorial Pattern Matching*, volume 644 of *Lecture Notes in Computer Science*, pages 185–192. Springer-Verlag, 1992.

[30] S. C. Chang and E. L. Lawler. Sublinear expected time approximate string matching and biological applications. *Algorithmica* **12** (1994) 327–344.

[31] A. Apostolico and M. Crochemore. String pattern matching for a deluge survival kit. In *Handbook of Massive Data Sets*, J. Abello, P.M. Pardalos and M.G.C. Resende, eds., Kluwer Academic Publishers, 2000) to appear.

[32] P. Weiner. Linear pattern matching algorithm. *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, Washington, D.C., 1973, 1–11.

[33] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. Algorithms*, 23(2):262–272, 1976.

[34] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M.T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoret. Comput. Sci.* **40** (1985) 31–55.

[35] M. Crochemore. Transducers and repetitions. *Theoret. Comput. Sci.*, 45(1):63–86, 1986.

[36] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[37] G. Gonnet, R. Baeza-Yates, and T. Snider. Lexicographical indices for text: inverted files vs. PAT trees. Technical report OED-91-01, Centre for the New OED, University of Waterloo, 1991.

[38] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. Proc. 2nd Ann. Intern. Computing and Combinatorics Conference, Lecture Notes in Computer Science number1090, Springer-Verlag, 1996, 219–230.

[39] J. Kärkkäinen. Suffix cactus: a cross between suffix tree and suffix array. Proc. 6th Ann. Symp. on Combinatorial Pattern Matching, Lecture Notes in Computer Science number937, Springer-Verlag, 1995, 191–204.

[40] P. Ferragina and R. Grossi. A fully-dynamic data structure for external substring search. *Proceedings of the 27th ACM Symposium on the Theory of Computing*, ACM Press, 1995.

[41] A. Anderson and S. Nilson. Efficient implementation of suffix trees. *Softw. Pract. Exp.*, 25:129–141, 1995.

[42] R. W. Irving. Suffix binary search trees. Technical report 1995-7, University of Glasgow, 1995.

[43] M. Crochemore and R. Vérin. On compact directed acyclic word graphs. Edited by J. Mycielski, G. Rozenberg, and A. Salomaa. *Structures in Logic and Computer Science*. Lecture Notes in Computer Science number1261. Springer-Verlag, 1997, 192–211.

[44] A. Blumer, A. Ehrenfeucht, and D. Haussler. Average sizes of suffix trees and DAWGS. *Discrete Applied Mathematics* **24** (1989) 37–45.

[45] M. Raffinot. Asymptotic estimation of the average number of terminal states in dawgs. *Proceedings of the 4th South American Workshop on String Processing*, Carleton University Press, 1997, 140–148.

[46] S. Kurtz. Reducing the space requirement of suffix trees. Technical report 98-03, University Bielefeld, 1998.

[47] J. Holub. Personal communication, 1999.

[48] M. Balík. Searching substrings. Technical report DC-PSR-2000-02, Czech Technical University, 2000.

[49] U. Manber and R. Baeza-Yates. An algorithm for string matching with a sequence of don't cares. *Inform. Process. Lett.* **37**:3 (1991) 133–136.

[50] G. Navarro. Indexing methods for approximate string matching. Technical report, University of Chile, 2000.

[51] N. El-Mabrouk and F. Lisacek. Very fast identification of RNA motifs in genomic DNA. Application to tRNA search in the yeast genome. J. Mol. Biol. **264** (1996) 46-55.

[52] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Inf. Proc. Letters*, 12:244–250, 1981.

[53] A. Apostolico et F. P. Preparata. Optimal off-line detection of repetitions in a string. *Theoret. Comput. Sci.*, 22(3):297–315, 1983.

[54] M. G. Main et R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. Algorithms*, 5(3):422–432, 1984.

[55] R. Kolpakov et G. Kucherov. Finding maximal repetitions in a word in linear time, *Symposium on Foundations of Computer Science* (FOCS), New-York (USA), 596-604, IEEE Computer Society, 1999.

[56] J. Stoye and D. Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. In M. Farach-Colton, editor, *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, number 1448 in Lecture Notes in Computer Science, pages 140–152, Piscataway, NJ, 1998. Springer-Verlag, Berlin.

[57] B. Charlesworth, P. Sniegowski, and W. Stephan. The evolutionary dynamics of repetitive DNA in eukaryotes. *Nature*, 371:215–220, 1994.

[58] M.-F. Sagot and E. W. Myers. Identifying satellites in nucleic acid sequences. In S. Istrail, P. Pevzner, and M. Waterman, editors, *RECOMB'98. Proceedings of Second Annual International Conference on Computational Molecular Biology*, pages 234–242. ACM Press, 1998.

[59] B. Clift, D. Haussler, R. McConnell, T. D. Schneider, and G. D. Stormo. Sequence landscapes. *Nucleic Acids Res.*, 14:141–158, 1986.

[60] A. Milosavljevic and J. Jurka. Discovering simple DNA sequences by the algorithmic significance method. *Comput. Appl. Biosci.*, 9:407–411, 1993.

[61] O. Delgrange. *Un algorithme rapide pour une compression modulaire optimale. Application à l'analyse de séquences génétiques.* PhD thesis, 1997. Thèse de doctorat - Université de Lille I.

[62] V. Fischetti, G. Landau, J. Schmidt, and P. Sellers. Identifying periodic occurrences of a template with applications to protein structure. In Z. Galil A. Apostolico, M. Crochemore and U. Manber, editors, *Combinatorial Pattern Matching*, volume 644 of *Lecture Notes in Computer Science*, pages 111–120. Springer-Verlag, 1992.

[63] E. Rivals and O. Delgrange. A first step toward chromosome analysis by compression algorithms. In N. G. Bourbakis, editor, *First International IEEE Symposium on Intelligence in Neural and Biological Systems*, pages 233–239. IEEE Computer Society Press, 1995.

[64] P. Bieganski, J. Riedl, J. V. Carlis, and E.M. Retzel. Generalized suffix trees for biological sequence data: applications and implementations. In *Proc. of the 27th Hawai Int. Conf. on Systems Sci.*, pages 35–44. IEEE Computer Society Press, 1994.

[65] L. C. K. Hui. Color set size problem with applications to string matching. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Combinatorial Pattern Matching*, volume 644 of *Lecture Notes in Computer Science*, pages 230–243. Springer-Verlag, 1992.

[66] L. Marsan and M.-F. Sagot. Extracting structured motifs using a suffix tree – algorithms and application to promoter consensus identification. In S. Istrail, P. Pevzner, and M. Waterman, editors, *RECOMB'00. Proceedings of Fourth Annual International Conference on Computational Molecular Biology.* ACM Press, 2000.

[67] J. van Helden, A. F. Rios, and J. Collado-Vides. Discovering regulatory elements in non-coding sequences by analysis of spaced dyads. *Nucleic Acids Res.*, 28:1808–1818, 2000.

[68] I. Jonassen, J. F. Collins, and D. G. Higgins. Finding flexible patterns in unaligned protein sequences. *Protein Science*, 4:1587–1595, 1995.

[69] I. Jonassen. Efficient discovery of conserved patterns using a pattern graph. *Comput. Appl. Biosci.*, 13:509–522, 1997.

[70] Y. M. Fraenkel, Y. Mandel, D. Friedberg, and H. Margalit. Identification of common motifs in unaligned DNA sequences: application to *Escherichia coli lrp* regulon. *Comput. Appl. Biosci.*, 11:379–387, 1995.

[71] A. Klingenhoff, K. Frech, K. Quandt, and T. Werner. Functional promoter modules can be detected by formal models independent of overall nucleotide sequence similarity. *Bioinformatics 1*, 15:180–186, 1999.

[72] A. Vanet, L. Marsan, A. Labigne, and M.-F. Sagot. Inferring regulatory elements from a whole genome. An analysis of the σ^{80} family of promoter signals. *J. Mol. Biol.*, 297:335–353, 2000.

[73] A. Vanet, L. Marsan, and M.-F. Sagot. Promoter sequences and algorithmical methods for identifying them. *Research in Microbiology*, 150:779–799, 1999.