Longest Motifs with a Functionally Equivalent Central Block

Maxime Crochemore, Raffaele Giancarlo, Marie-France Sagot

Longest Motifs with a Functionally Equivalent Central Block

Maxime Crochemore[1,2⋆], Raffaele Giancarlo[3⋆⋆], Marie-France Sagot[4,2⋆⋆⋆]

[1] Institut Gaspard-Monge, University of Marne-la-Vallée,
77454 Marne-la-Vallée CEDEX 2, France
maxime.crochemore@univ-mlv.fr
[2] Department of Computer Science, King's College London
London WC2R 2LS, England
[3] Dipartimento di Matematica ed Applicazioni
Universitá di Palermo,
Via Archirafi 34, 90123 Palermo, Italy
raffaele@altair.math.unipa.it
[4] Inria Rhône-Alpes, Laboratoire de Biométrie et Biologie Évolutive,
Université Claude Bernard, 69622 Villeurbanne cedex, France
Marie-France.Sagot@inria.fr

Abstract. This paper presents a generalization of the notion of longest repeats with a block of k don't care symbols, introduced by [8] for k fixed, to longest motifs composed of three parts: a first and last that parameterize match (that is, match via some renaming, initially unknown), and a functionally equivalent central (sub)block. Such three-part motifs are called *longest (one) (sub)block*. Different types of functional equivalence, and thus of matching criteria for the central block are considered, which include as a subcase the one treated in [8] and extend to the case of regular expressions with no Kleene closure nor complement operation. We show that a single general algorithmic tool that is a non-trivial extension of the ideas introduced in [8] can handle all the various kinds of longest block motifs defined in this paper. The algorithm complexity is, in all cases, in $O(n \log n)$.

1 Introduction

Crochemore et al. [8] have recently introduced and studied the notion of longest repeat with a block of k don't care symbols, where k is fixed. These are factors of the form $V \diamond^k W$ that appear repeated in a string X, where $\diamond^k$ is a region whose content can be arbitrary, *i.e.*, we do not care about its content. In a sense, those repeats are the simplest type of structured motifs that can appear in a string. In fact, their work is motivated by the study of this important special case of motifs and has some relation with previous work on repeats with bounded gaps [5, 12]. More in general, the term *motif* [9] is often used in biology to describe similar functional components that several biological sequences may have in common. It can also be used to describe any collection of similar factors of a longer sequence. In nature, many motifs are *composite*, *i.e.*, they are composed of conserved parts separated by random regions of variable lengths. By now, the literature on motif discovery is very rich [4], although a completely satisfactory algorithmic solution has not been reached yet.

Even richer (see [16, 17, 15]) is the literature on the characterization and detection of regularities in strings, where the object of study ranges from identification of periodic parts to identification of parts that simply appear more than once. Baker [2, 3] has contributed the notion of parameterized strings and has given several algorithms that find maximal repeated factors in a string that p-match, *i.e.*, they are identical up to a renaming of the symbols. Parameterized strings are a successful tool for the identification of duplicated parts of code in large software systems. These are pieces of code that

⋆ Partially supported by CNRS, Wellcome Foundation, and Nato grants.
⋆⋆ Partially supported by Italian MIUR grants PRIN "Metodi Combinatori ed Algoritmici per la Scoperta di Patterns in Biosequenze" and FIRB "Bioinformatica per la Genomica e La Proteomica". Additional support provided by CNRS, France, by means of a Visiting Fellowship to Institut Gaspard Monge.
⋆⋆⋆ Partially supported by French Programme BioInformatique Inter EPST, Wellcome Foundation, Royal Society and Nato grants.

are identical, except for a consistent renaming of variables. Motivated by practical as well as theoretical considerations, Amir et al. [1] have investigated the notion of function matching that incorporates parameterized strings as a special case.

Such investigations of factors that are "similar" according to a well defined correspondence hint at the existence of meaningful regularities in strings, such as motifs, that may not be captured by standard notions of equality. In particular, the presence of large duplications in software systems as well the experimental results in Structural Biology [13, 14, 22] indicating that, to some extent, there are equivalent amino acids, both provide examples where a motif may be more subtle than identical pieces interspersed with don't care symbols.

In this paper, we make a first substantial step in studying a new notion of motifs, where equality of strings is replaced by more general "equivalence" rules. We consider the simplest of such structured motifs, *i.e.*, the ones of the form $V \diamond^k W$, with k fixed, which we refer to as *block motifs*. One important point in this study is that the notation $\diamond^k$, usually indicating a don't care part, assumes here an entirely new meaning. Indeed, $\diamond^k$ is a place holder stating that, for two strings described by the motif, the portion of each string going from position $|V| + 1$ to $|V| + k - 1$, referred to as the *central part*, must match according to a specified set of rules. To illustrate this notion, consider $ab \diamond^2 ab$ and the rule stating that any two strings described by the motif must have their central part identical, except for a renaming of symbols. For instance, $abxyab$ and $ababab$ are both described by $ab\diamond^k ab$ and the given rule, since there is a one-to-one correspondence between xy and ab. Notions associated with the example and the intuition just given are formalized in Section 3, where the central part $\diamond^k$ is specified by a set of matching criteria, all related to parametrized strings and function matching. Moreover, in Section 6, we extend our approach to a central part of a motif being a fixed regular expression, containing no Kleene closure or complement operation. Our main contribution here is a formal treatment of this new type of motifs, resulting in conditions under which their definition is sound.

At the algorithmic level, our main contribution is to provide one general algorithm that extracts all longest block motifs, present in a string of length n, in $O(n \log n)$ time. Indeed, for each of the matching criteria for the central part presented in Sections 3 and 6, the general algorithm specializes to find that type of motif by simply defininining a new lexicographic order relation on strings. Even more remarkably, we show that the techniques in [8], in conjunction with some additional ideas presented here, can be naturally extended to yield a general algorithmic tool to discover more subtle repeated patterns in a string.

2 Parameterized Strings and Matching Via Functions

We start by recalling some basic definitions from the seminal work by Brenda Baker on Parameterized Strings [2, 3]. Let Σ and Π be two alphabets, referred to as *constants* and *parameters*, respectively. A *p-string* X is a string over the union of these two alphabets. A p-string is therefore just like any string, except that some symbols are parameters. In what follows, for illustrative purposes, let $\Sigma = \{a, b\}$ and $\Pi = \{u, v, x, y\}$. Baker gave a definition of matching for p-strings, which reduces to the following:

Definition 1. Two p-strings X and Y *p-match* if and only if X can be transformed into Y by applying a bijection G from the symbols of X to the symbols of Y, such that G is the identity on the constants.

Example 2. $X = abuvabuvu$ and $Y = abxyabxyx$ are a p-match, with G as the bijection, where $G(u) = x$ and $G(v) = y$.

For ease of reference, let $\Sigma_1 = \Sigma \cup \Pi$. From now on, we refer to p-strings simply as strings over the alphabet Σ_1 and, except otherwise stated, we assume that the notion of match coincides with that of p-match. We refer to the usual notion of match for strings as exact match. In that case, Σ_1 is treated as a set of constants. Moreover, we refer to bijections over Σ_1 as renaming functions. We also use the term prefix, suffix and factor in the usual way, *i.e.*, the i-th suffix of X is $x_i x_{i+1} \cdots x_n$, where n is the length of the string. In what follows, let $\overline{X}$ denote its reverse, *i.e.*, $x_n \cdots x_1$.

We also need to recall the definition of parameterized suffix tree, denoted p-suffix tree, also due to Baker [2, 3]. Its definition is based, among other things, on a suitable transformation of suffixes and prefixes of a string so that, when they match, they can share a path in a Patricia Tree. Indeed, consider the string $Y = uuuvvv$, made only of the parameters u and v. Notice that uuu and vvv are a match, and therefore they should share a path, when the suffixes of the string

are "stored" in a Compacted Trie. That would not be possible if the Compacted Trie were over the alphabet Σ_1. We now briefly discuss the ideas behind this transformation. Consider a new alphabet $\Sigma_2 = \Sigma \bigcup N$, where N is the set of nonnegative integers.

Definition 3. *Let prev be a transformation function on a string X, operating as follows. For each parameter, its first occurrence is replaced by 0 and each successive occurrence is represented by its distance, along the string, to the previous occurrence. Constants are left unchanged. Moreover, given a string X, we define its prev representation to be the string prev(X), over the alphabet Σ_2.*

Example 4. The *prev* function basically substitutes parameters with integers, leaving the constants unchanged, i.e, it transforms strings over Σ_1 into strings over Σ_2. For example, *prev*$(abxyxzaaya) = ab0020aa5a$.

The notion of match on strings corresponds to equality in their *prev* representation [2, 3]:

Fact 1 *Two strings X and Y are a match if and only if prev(X) =prev(Y). Moreover, these two strings are a match if and only if $\overline{X}$ and $\overline{Y}$ are.*

Notice that the *prev* representation of two strings tells us nothing about which factors, in each string, are a p-match. For instance, consider $abxyxzaaya$ and $zzzztzwaata$. Factors $xyxzaaya$ and $ztzwaaaata$ match, but that cannot be directly inferred from the *prev* representation of the two full strings.

Definition 5. *Let X be a string that ends with a unique endmarker symbol. A parameterized suffix tree for X (p-suffix tree for short) is a compacted trie storing all suffixes of X, via their prev representation.*

Definition 5 is sound in the sense that all factors of X are represented in the p-suffix tree (that follows because each factor is prefix of some suffix). Even more importantly, matching factors share a path in the tree. Indeed, consider two factors that match. Assume that they are of length m. Certainly they are prefixes of two suffixes of X. When represented via the *prev* function, these two suffixes must have equal prefixes of length at least m (by Fact 1). Therefore, the two factors must share a path in the p-suffix tree. Consider again $Y = uuuvvv$. Notice that *prev*$(uuuvvv) = 012012$ and that *prev*$(vvv) = 012$, so uuu and vvv can share a path in the p-suffix tree.

For later use, we also need to define a lexicographic order relation on strings, via their *prev* representation. It reduces to the usual definition when the string has no parameters. Consider the alphabet Σ_2 and let $\leq_2$ denote the standard lexicographic order relation for strings over a fixed alphabet: the subscript indicates to which alphabet the relation refers to.

Definition 6. *Let X and Y be two strings. We say that X is lexicographically smaller than Y if and only if prev$(X) \leq_2$ prev(Y). We indicate such a relation via $\leq_2$.*

In what follows, we need also another type of function that, somewhat improperly, we define as a table:

Definition 7. *A Table T has domain Σ_1 and range the power set of Σ_1. Fix two tables T and T' and two strings X and Y. X table matches Y, for short t-matches, via the two tables T and T', if and only if $y_i \in T(x_i)$ and $x_i \in T'(y_i)$, $1 \leq i \leq n$. For later reference, $X \rightarrow_T Y$ indicates that X can be transformed into Y via T.*

Example 8. Let $T(a) = \{a, u\}$, $T(b) = \{x, v, y\}$, $T'(a) = T'(u) = \{a\}$ and $T'(x) = T'(v) = T'(y) = \{b\}$. Then $X = aaabbb$ and $Y = auaxvy$ t-match.

A substantial difference between tables and p-strings is that, for tables, all symbols in Σ_1 are treated as parameters and the correspondence is fixed once and for all by the table.

For arbitrary tables, t-matching is not an equivalence relation. In fact, although symmetry is implied by the definition, neither reflexivity nor transitivity are. This depends very much on the type of tables for which the match holds. Notice also that t-matching incorporates the notion of match with don't care, when both tables correspond to the one that assigns the entire alphabet to any symbol of the alphabet. We refer to this latter table as the *don't care* table. However, for particular families of tables, the notion of t-match is that of a match:

Fact 2 *X and Y match* **and** *t-match if and only if both tables are many-to-one functions. In particular, the two tables can be transformed into renaming functions.*

Proof. The only non-trivial part is the one about many-to-one functions. Let T and T' be the two many-to-one functions by means of which the t-match is verified.

Assume that for no pair of distinct symbols a and b, appearing in X, we have that $T(a) = T(b)$. Then, we can transform T into a renaming function and X and Y would match. Indeed, for each symbol a appearing in X, $T(a)$ is assigned to it. That causes no conflict. Now, since all symbols of the alphabet appearing in X are assigned one-to-one, we can arbitrarily assign the symbols not in X also one-to-one. The result is a renaming function implying that X and Y match.

Now, assume that there exist at least two symbols a and b in X such that $T(a) = T(b)$. Since a and b both occur in X and $T(a)$ occurs at the corresponding positions of Y, we have that T' cannot be a function. Indeed, $T'(T(a))$ must have both value a and b. Therefore, both T and T' must be renaming functions, implying that X and Y match. □

3 Functions and Block Motifs

We now investigate the notion of block motif, which was termed repeat with a block of don't cares in [8], in conjunction with that of renaming functions and tables. The basic idea is that a block motif is a concise way of expressing a set of factors which appear in X and which are all related by the fact that they are "identical", except for a well specified "central part". For ordinary strings, it is possible to define such a notion because "identity" is obviously an equivalence relation and therefore a block motif can be seen as a representative item from its equivalence class, with the central part excluded. We follow this intuition and the construction associated to it, which is immediate for ordinary strings. To this end, we need to define the notion of repetition and motif. They are generalizations of the corresponding ones given in [8] for ordinary strings.

Fix a family of tables $\mathcal{T}$ and an integer k, $0 \leq k \leq n$, and consider also a family of renaming functions.

Definition 9. Let Y be a factor of X. Y is a *general k-repeat* if and only if the following conditions hold: (a) Y can be written as VQW, V and W both non-empty and $|Q| = k$ (b) there exists another factor Z of X, two renaming functions F and G and two tables in $\mathcal{T}$, for which $Z = F(V)Q'G(W)$ and Q and Q' t-match, via the two tables.

Definition 10. *Let $R(k, i, j)$ be the following binary relation on strings of length m, with $j < m$ and $k = j - i + 1$: $ZR(k, i, j)Y$ if and only if $(z_1 z_2 \cdots z_{i-1})$, $(z_{j+1} \cdots z_m)$ and $(y_1 x_2 \cdots y_{i-1})$, $(y_{j+1} \cdots y_m)$ match, respectively, while $(z_i \cdots z_j)$ and $(y_i \cdots y_j)$ t-match via two possibly distinct tables in $\mathcal{T}$, $1 < i \leq j < m$.*

Definition 11. *Given a string X, consider a factor Y, of length m, and assume that it is a k-repeat. Fix i and j as in Definition 10 and consider all factors Z of X such that $YR(k, i, j)Z$. Assume that $R(k, i, j)$ is an equivalence relation. Then, for each class with at least two elements, a block motif is any arbitrarily chosen factor in that class, say Y. As for standard strings, the block motif can be written as $y_1 y_2 \cdots y_{i-1} \diamond^k y_{j+1} \cdots y_m$, once it is understood that $\diamond^k$ is a place holder specifying a central part of the motif and that the matching criterion for that part is given by the family of tables.*

We point out that, in general, $R(k, i, j)$ need not be an equivalence relation. Later we consider conditions under which it is indeed an equivalence relation, therefore allowing us to define block motifs.

Example 12. Restrict the family of tables to be only the don't care table. Let $Z = abvvva$ and $Y = abxxya$, and consider $ZR(2, 3, 4)Y$ with the identity function for the prefix ab and $G(v) = y$ and $G(a) = a$ for the suffix of length 2. Moreover, consider $X = YZ$. Then, $ab \diamond^2 va$ is a block motif. Also $ab \diamond^2 ya$ is a block motif, but it is equivalent to the other one, given the choices made about the family of tables and the fact that we are using a notion of match via renaming.

We now investigate the types of table families that allow us to properly define block motifs. As it is clear from Example 12, the notion of block motifs, as defined in [8], is a special case of the ones defined here: it satisfies Definition 11, when (A) we restrict the family of tables in Definition 10 to consist only of the don't care table; and (B) the renaming

functions are restricted to be the identity function. It is also clear that the family of all tables yields the same notion of block motif as the one with only the don't care table. However, exclusion of only the don't care table is not enough to obtain a proper definition of block motifs:

Lemma 13. *Restrict the family of tables to consist of all tables, except the don't care one. For any alphabet of size at least two, there exists an infinite set of values of $k \geq k_0$, k_0 depending on the alphabet size, such that $R(k,i,j)$ is not an equivalence relation.*

Proof. All we have to do is to produce three strings such that Z_1 and Z_2 t-match, Z_2 and Z_3 t-match, with tables other than the don't care table, but Z_1 and Z_3 will t-match only with the don't care table. These strings can be used as "the middle" part of three other strings, for which transitivity fails in R. Consider a binary alphabet first. Let $Z_1 = aabb$, $Z_2 = abaa$ and $Z_3 = abab$. It is straightforward to verify that Z_1 and Z_2 t-match, Z_2 and Z_3 t-match, with tables other than the don't care table. But Z_1 and Z_3 t-match *only* with the don't care table. Obviously the same statements will hold for any three strings which are powers of Z_1, Z_2 and Z_3, respectively.

Assume that the alphabet has three symbols. We add an appropriate number of symbols to Z_1, Z_2 and Z_3, so that we can draw conclusions identical to the binary case. Let $Z_1 = aabb|aacc|bbcc$, $Z_2 = abaa|acaa|bcbb$ and $Z_3 = abab|acac|bcbc$. We have divided the strings in pieces for ease of reference. The first piece in each string is as the binary case. The remaining pieces have the function to constraint at most two symbols to match the entire alphabet, while the third one will be so constrained only when we consider Z_1 and Z_3. The construction generalizes to arbitrary alphabets. □

Fortunately, there are easily checkable sufficient conditions ensuring that the family of tables guarantees R to be an equivalence relation, as we show next.

Definition 14. *Consider two tables T and T'. Let their composition, denoted $\circ$, be $T \circ T'(a) = \bigcup_{c \in T'(a)} T(c)$, for each symbol a in the alphabet. $\mathcal{T}$ is closed under composition if and only if, for any two tables in the family, their composition is a table in the family.*

Definition 15. *A table T contains a table T' if and only if $T'(a) \subseteq T(a)$, for each symbol a in the alphabet.*

Lemma 16. *Assume that $\mathcal{T}$ is closed under composition and that there exists a table in $\mathcal{T}$ containing the identity table. Then R is an equivalence relation.*

Proof. Since there exists a table in $\mathcal{T}$ containing the identity table, then R is certainly reflexive. Notice that R is symmetric: (a) it is explicitly required by the definition of t-matching and (b) it obviously holds for renaming functions. We need to show that R is transitive. Consider three strings Z_1, Z_2 and Z_3, such that $Z_1 R(k,i,j) Z_2$ and $Z_2 R(k,i,j) Z_3$. Notice that the match relation is transitive, by definition of a renaming function. When restricted to the family of tables in $\mathcal{T}$, also the table match relation is transitive, since by assumption the family is closed under composition. □

In order to obtain a partial inverse of the previous Lemma, we need some definitions.

Definition 17. *Assume that R is an equivalence relation for strings in Σ^m. For each equivalence class C containing at least three strings, we define its matching graph G_C as follows. Assign a vertex to each string and connect each vertex to any other vertex via a directed edge. Label the edge (u,v) with a table T, denoted $T_{u,v}$, by means of which the string X associated to u and the string Y associated to v are such that $X \rightarrow_T Y$.*

Lemma 18. *Assume that R is an equivalence relation. Then, there is a table in $\mathcal{T}$ containing the identity table. Moreover, for each equivalence class C with at least three strings the following holds. For any three vertices, u, v and w in G_C, $T_{u,w}(a) \cap (T_{v,w} \circ T_{u,v}(a)) \neq \emptyset$, for each symbol in the string associated to u.*

Proof. There must be a table in $\mathcal{T}$ containing the identity table, else R could not be reflexive.

For the second part of the lemma, let X, Y and Z be the three strings associated to u, v and w. Consider $x_i = a$. Then, $y_i = b \in T_{u,v}(a)$. But $z_i = c \in T_{v,w}(b)$. That is, $c \in T_{v,w} \circ T_{u,v}(a)$. But $c \in T_{u,w}(a)$. □

We now consider some interesting special classes of table functions, in particular four of them, for which we can define block motifs. Let $\mathcal{T}_\diamond$ consist only of the don't care table. Let $\mathcal{T}_r$ and $\mathcal{T}_m$ consist of renaming functions and many-to-one functions, respectively. In order to define the fourth family, we need some remarks.

The use of tables for the middle part of a block motif allows us to specify simple substitution rules a bit more relaxed than renaming functions. We discuss one of them. Partition the alphabet into classes and let $\mathcal{P}$ denote the corresponding partition. Define a *partition table* $\mathcal{T}_\mathcal{P}$ that assigns to each symbol the class it belongs to. For instance, fix two characters in the alphabet, say a and b. Consider the table, denoted for short $T_{a,b}$, that assigns $\{a, b\}$ to both a and b and the symbol itself to the remaining characters. In a sense, $\mathcal{T}_\mathcal{P}$ formalizes the notion of groups of characters being interchangeable, or equivalent. Those situations arise in practice (see for instance [6, 11, 13, 14, 19, 21, 22]). Indeed, in order to simplify the study of protein folding, there have been many efforts to partition the set of amino acids into classes of interchangeable ones. The partition to choose is situation dependent and probably no unique or optimal partition exists. However, a few experimental studies have shown that protein sequence analysis, and even folding, is reliable when, in the original amino acid sequence, one substitutes each character with its class. In many cases, experiments show that the new sequence so obtained reliably represents properties of the original sequence (see for instance [14]). Of course, in the best of all possible worlds, one would like to use a relation among amino acids that is realistic, rather than mathematically and computationally pleasing. In a sense, the experimental results mentioned earlier for equivalent classes of amino acids, show however that those classes are a good first approximation to more realistic relations.

We need the following:

Fact 3 *Let X and Y be two strings of equal length. We have that $x_i \in \mathcal{T}_\mathcal{P}(y_i)$ if and only if $y_i \in \mathcal{T}_\mathcal{P}(x_i)$.*

Proof. Assume that $x_i \in \mathcal{T}_\mathcal{P}(y_i)$, but the converse is not true. So, $\mathcal{T}_\mathcal{P}(y_i) \neq \mathcal{T}_\mathcal{P}(x_i)$, but since x_i is in both, we have that $\mathcal{P}$ is not a partition. □

Let the fourth family of tables consist of only $\mathcal{T}_\mathcal{P}$, for some given partition $\mathcal{P}$ of the alphabet Σ_1. Notice that Fact 3 ensures that the definition of t-matching is well posed in this case.

Lemma 19. *Pick any one of $\mathcal{T}_\diamond, \mathcal{T}_r, \mathcal{T}_m$ or $\mathcal{T}_\mathcal{P}$ and consider the relation R in Definition 10 for the chosen family. R is an equivalence relation. In particular, when the chosen family is $\mathcal{T}_m$, R is the same relation as that for $\mathcal{T}_r$.*

Proof. Notice that, all four families contain the identity table. So, by Lemma 16, all that remains to show is that those four families are closed under composition. That is well known for the first three. As for $\mathcal{T}_\mathcal{P}$, simply notice that it is its own composition.

For the second part of the lemma, with reference to Definition 10, consider $z_i \cdots z_j$ and $y_i \cdots y_j$ and apply Fact 2 to them. □

Example 20. Fix the family of tables to be one-to-one functions. Consider $X = YZ$, where Y and Z are again the strings in Example 12. Then, $ab \diamond^2 va$ and $ab \diamond^2 ya$ are block motifs representing the same class, the one consisting of Y and Z. We can pick any one of the two, since they are equivalent. Notice that the rule for the central part states that the corresponding region for two strings described by the motifs must be each a renaming of the other.

Example 21. Fix the family of tables to be $T_{a,b}$. Let $Z = cdccdacdc$ and $Y = cdccdbcdc$. Let $X = ZY$. Then $cdc \diamond^k cdc$ is a block motif, representing both Y and Z. Again, the rule for the central part states that the corresponding region for two strings described by the motif must be identical, except that a and b can be treated as the same character.

4 Longest Block Motifs with a Fixed Partition Table

We now give an algorithm that finds all longest block motifs in a string, when we use a partition table, known and fixed once and for all. The algorithm is a non-trivial generalization of the corresponding one in [8]. In fact, we show that the main techniques used there, and that we nickname as *the two-tree trick*, is a very powerful tool to extract longest block motifs in various settings, when used in conjunction with the algorithmic ideas presented in this Section.

Indeed, a verbatim application of the two-tree trick would work on the p-suffix trees for the string and its reverse. Unfortunately, that turns out to be not enough in our setting. We need to construct a tree somewhat different than a p-suffix tree, which we refer to as a p-suffix tree on a mixed alphabet. Using this latter tree, the techniques in [8] can be extended. Moreover, due to the generality of the algorithm constructing this novel version of the p-suffix tree, all the techniques we discuss in this Section extend to the other three types of block motifs defined in Section 3, as it is discussed in Sections 5 and 6.

For each class in $\mathcal{P}$, select a representative. The representatives give a reduced alphabet Σ_3. For any string Y, let $\hat{Y}$ be its corresponding string on the new alphabet, obtained by replacing each symbol in Y with its representative. In what follows, for our examples, we choose $T_{a,b}$, with a as representative. Consider a string X and assume that it has block motif $V \diamond^k W$, with respect to table $\mathcal{T}_{\mathcal{P}}$. We recall that $V \diamond^k W$ is a shorthand notation for the fact that strings in the class (a) t-match in the positions corresponding to the central part and, (b) they (parametric) match in the positions corresponding to V and W. We are interested in finding all longest block motifs.

Consider a Patricia Tree T, storing a set of strings. Let Y be a string. The locus u of Y in T, if it exists, is the node such that Y matches the string corresponding to the path from the root of T to u. Notice that when T is a p-suffix tree, then *prev*(Y) must be the string on the path from the root to u. For standard strings, the definition of locus reduces to the usual one. With those differences in mind, one can also define in the usual way the notion of contracted and extended locus of a string. Moreover, given a node u, let $d(u)$ be the length of the string of which u is locus.

4.1 A p-suffix Tree on a Mixed Alphabet

Definition 22. *The modified prev representation of a string Y, denoted as mprev(Y), is defined as follows. If $|Y| \leq k$, then it is $\hat{Y}$. Else, it is $\hat{W}$prev(Z), where $Y = WZ$ and $|W| = k$. For instance, let $Y = abauuxx$, and $k = 3$. Then, its modified prev representation is mprev$(Y) = aaa0101$.*

Definition 23. *Let X be a string with a unique endmarker. Let T'_X be a Patricia Tree storing each suffix of X, via their mprev representation. That is, T'_X is like a p-suffix tree, but the initial part of each suffix is represented on the reduced alphabet. For instance, let $X = abbabbb$ and $k = 2$, the first suffix of X is stored as $aababbb$.*

Notice that T'_X has $O(n)$ nodes, since it has n leaves and each node has outdegree at least two. Moreover, each edge can be labelled with pointers to factors of X, as it is custumary for suffix trees [18]. That would allow us to use it for pattern matching, by resorting to some additional techniques from Baker [3]. So the total size of the tree is $O(n)$. We leave the details to the interested reader. In fact, as it will become clear later, we only need to build and use the topology of T'_X, since we do not use it for pattern matching.

We now show how to build T'_X in $O(n \log n)$ time. Let `BuildTree` be a procedure that takes as input the n suffixes of X and returns as output T'_X. The only primitive that the procedure needs to use is the check, in constant time, for the lexicographic order of two suffixes, according to a new order relation that we define. The check should also return the longest prefix the two suffixes have in common and which suffix is smaller than the other.

Definition 24. *Let Y and Z be two strings. We define a new order relation $Y \leq_m Z$ as follows. When $|Y| \leq k$, it must be $\hat{Y} \leq_3 \hat{U}$, where $|U| = |Y|$ is a prefix of Z. Assume that $|Y| > k$, and let $Z = US$ and $Y = RP$, with $|R| = |U| = k$. Then, it must be $\hat{R} <_3 \hat{U}$ or $\hat{R} = \hat{U}$ but $prev(P) \leq_2 prev(S)$. With a little abuse of notation, we can write $mprev(Y) \leq_m mprev(Z)$, when $Y \leq_m Z$.*

Given the suffix trees $T_{\hat{X}}$ [18] and the p-suffix tree T_X, assume that they have been processed to answer LCA queries in constant time [10, 20]. Then, it is easy to check, in constant time, the $\leq_m$ order of two suffixes of X, via two LCA queries in those trees. Moreover, that also gives us the length of the matching prefix. The details are omitted. We refer to such an operation as `compare(i, j)`, where i and j are the suffix numbers. It returns which one is smaller and the length of their common prefix.

Now, `BuildTree` works as follows. It simply builds the tree, without any labelling of the edges, as it is usual in Patricia Tries.

ALGORITHM BuildTree

1. Using `compare` and the $\leq_m$ relation, sort the suffixes of X with, say, Heapsort [7].
2. Process the sorted list $i_1, \cdots, i_n$ of suffixes in increasing order as follows:
   2.1 When the first suffix is processed, create a root and a leaf, push them in a stack in the order they are created. Label the leaf with i_1.
   2.2 Assume that we have processed the list up to i_g and that we are now processing i_{g+1}. Assume that on the stack we have the path from the root to leaf labeled i_g in the tree built so far, from bottom to top. Let it be $u_1, u_2, \cdots, u_s$.
      2.2.1 Using `compare` and the $\leq_m$ relation, find the longest prefix that i_g and i_{g+1} have in common. Let Z denote that prefix and d be its length.
      2.2.2 Pop elements from the stack until one finds two such that $d(u_i) \leq d < d(u_{i+1})$. Pop u_{i+1} from the stack. If $d(u_i) = d$, then u_i is the locus of Z in the tree built so far. Else, u_i and u_{i+1} are its contracted and extended locus, respectively. If u_i is the locus of Z, add a new leaf labeled i_{g+1} as offspring of u_i and push it on the stack. Else, create a new internal node u, as locus of Z, add it as offspring of u_i and make u_{i+1} an offspring of u. Moreover, add a new leaf labeled i_{g+1} as offspring of u and push the new created nodes on the stack, in the order in which they were created. We now have on the stack the path from the root to the leaf labeled i_{g+1}.

Lemma 25. *Tree T'_X can be correctly built in $O(n \log n)$ time.*

Proof. Proof of correctness is by induction, using the fact that the strings are processed in sorted order. When a new suffix is inserted, the only new internal node that needs to be created must be on the path from the root to the leaf representing the last inserted suffix. That path is on the stack and it is correctly updated. As for the time complexity, the construction of the suffix tree $T_{\hat{X}}$ takes $O(n)$ time [18], while that of the p-suffix tree T_X takes $O(n \log n)$ time [2, 3]. The time to preprocess those trees so that LCA queries can be answered in constant time is again $O(n)$ [10, 20]. When T'_X is actually built, it takes $O(n)$ time, since step (2.) processes each internal node only a constant number of times and there are a total of $O(n)$ nodes. So, the sorting step is the most expensive, but via `compare`, that can be done in $O(n \log n)$ time. □

4.2 The Algorithm

Consider the trees T'_X and $T_{\overline{X}}$, where the latter one is a p-suffix tree. For each leaf labeled i in $T_{\overline{X}}$, change its label to be $n + 2 - i$, so that whenever the left part of a block motif starts at i in $\overline{X}$, we have the position in X where the right part starts, including the central part. We refer to those positions as *twins*. Visit T'_X in preorder. Consider the two leaves $\ell_1 \in T'_X$ and $\ell_2 \in T_{\overline{X}}$, corresponding to a pair of twins. Assign to ℓ_2 the same preorder number as that of ℓ_1. Let $V \diamond^k W$ be a block motif and let i be one of its occurrences in X, *i.e.*, where it starts. In order to simplify our notation, we refer to such an occurrence via the preorder number of the leaf assigned to $i + |V| + 1$ in T'_X. From now on, we will simply be working with those preorder numbers. Indeed, given the tree we are in, we can recover the positions in X or $\overline{X}$ corresponding to the label at a leaf in constant time, by suitably keeping a set of tables. The details are as in [8]. Moreover, we can also recover the position where a block motif occurs, given the block motif and the preorder number assigned to the position. Given a tree T, let $L(v)$ be the list of labels assigned to the leaves in the subtree rooted at v. For the trees we are working with, those would be preorder numbers.

Definition 26. *We say that $V \diamond^k W$ is maximal if and only if extending any factor in the class, both to the left and to the right, results in the loss of at least one element in the class. That is, by extending the strings in the class, we can possibly get a new block motif, but its class does not contain that of $V \diamond^k W$.*

Example 27. Let $X = aabbaxxbxababyyayabbbuu$. Block motif $ab \diamond^2 xx$ is maximal. Indeed, it represents the class of factors $\{abbaxx, ababyy, abbbuu\}$. However, extending any of those factors both to the right and to the left results in a smaller class.

Lemma 28. *Consider a string X, its reverse, the trees $T_{\overline{X}}$ and T'_X. Assume that $V \diamond^k W$ is maximal. Pick any representative in the class, say VQW. Then $\overline{V}$ and mprev(QW) have a locus u in $T_{\overline{X}}$ and v in T'_X, respectively. Moreover, all the occurrences of $V \diamond^k W$ are in $L(u) \bigcap L(v)$. Conversely, pick two nodes u' and v', in $T_{\overline{X}}$ and T'_X, respectively. Assume that there are at least two labels i and j in $L(u') \bigcap L(v')$ such that $LCA(i,j) = u'$ and $LCA(i,j) = v'$, in $T_{\overline{X}}$ and T'_X, respectively. Assume also that $d(v') > k$. Then, they are occurrences of a maximal block motif.*

Proof. Assume that $V \diamond^k W$ is maximal and that $\overline{V}$ has an extended locus in $T_{\overline{X}}$, but not a locus. Pick a factor of X in the class and, for ease of notation, let it be VQW. Since $\overline{V}$ appears in $\overline{X}$ and such a string has no proper locus in $T_{\overline{X}}$, we have that all factors of $\overline{X}$ matching $\overline{V}$ will also match if they are extended to the right by one character in $\overline{X}$. But, by Fact 1, we have that all factors of X matching V can be extended to the left by one character and still match. But then, we can take all factors in the class $V \diamond^k W$ and extend them to the left without losing any element in the class. The same conclusion can be derived for QW, with an analogous reasoning. This contradicts the maximality of the block motif. To complete this part of the lemma, notice that V must match the prefix of any other factor of X in the class $V \diamond^k W$. Analogous considerations hold for *QW*. Therefore, all occurrences of $V \diamond^k W$ must be both in $L(u)$ and $L(v)$.

Conversely, pick two nodes u' and v', in $T_{\overline{X}}$ and T'_X, respectively. Let $d(v') > k$. Assume that there are at least two labels i and j in $L(u') \bigcap L(v')$ such that $LCA(i,j) = u'$ and $LCA(i,j) = v'$, in $T_{\overline{X}}$ and T'_X, respectively. Let Z be the string, the reverse of which has locus u' in $T_{\overline{X}}$, and let P, $|P| = k$, and M be two strings such that *mprev*(PM) has locus v' in T'_X. Notice that since $d(v') > k$, a string of the claimed form must exist. Assume also that ZPM occurs at i in X. Then, $Z \diamond^k M$ must be a maximal block motif. Indeed, it also occurs at j and, if extended both to the left and to the right, i and j cannot be both occurrences of that extension. □

We also need the following:

Lemma 29. *Consider an internal node v in $T_{\overline{X}}$ and two of its offsprings, say, v_1 and v_2. Let $j_1, j_2, \cdots, j_m$ be the sorted list of labels assigned to the leaves in the subtree rooted at v_1 and let i be a label assigned to any leaf in v_2. Let g be the first index such that $j_g \leq i$. Similarly, let c be the first index such that $i \leq j_c$. The maximal block motif of maximum length that i forms with $j_1, j_2, \cdots, j_m$ is either with j_g, if it exists, or with j_c, if it exists, provided that either $d(LCA(i, j_g)) > k$ or $d(LCA(i, j_c)) > k$ and the LCA is computed on T'_X.*

Proof. Notice that, by definition of preorder visit of a tree, the sequence given by the depths of the LCA, in T'_X, of i with $j_c, \cdots, j_m$, respectively, is non-increasing. Now let $u = LCA(i, j_c)$ in T'_X and assume that $d(u) > k$. We know that $LCA(i, j_1) = v$ in $T_{\overline{X}}$. So, applying Lemma 28 to these two nodes, we have a maximal block motif. The length of the block motif is at least as long as that of any other possibly given by $j_{c+1}, \cdots, j_m$.

An analogous reasoning holds with $j_1, \cdots, j_g$, except that now the sequence of depths of the LCA's, in T'_X, of i with $j_1, \cdots, j_g$ is non-decreasing.

Therefore, the longest block motif must be with either j_g or j_{g+1}. □

We now present the algorithm.

ALGORITHM LM

1. Build $T_{\overline{X}}$ and T'_X. Visit T'_X in preorder and establish a correspondence between the preorder numbers of the leaves in T'_X and the leaves in $T_{\overline{X}}$. Transform $T_{\overline{X}}$ into a binary suffix tree $\mathcal{B}$ (see [8]);
2. Visit $\mathcal{B}$ bottom up and, at each node, merge the sorted lists of the labels (preorder numbers in T'_X) associated to the leaves in the subtrees rooted at the children. Let these lists be $\mathcal{A}_1$ and $\mathcal{A}_2$ and assume that $|\mathcal{A}_1| \leq |\mathcal{A}_2|$. Merge $\mathcal{A}_1$ into $\mathcal{A}_2$. Any time an element i of the first list is inserted in the proper place in the other, *e.g.*, j_g and j_c in Lemma 29 are identified, we only need to check for two possibly new longest maximal block motifs that i can generate. While processing the nodes in the tree, we keep track of the longest maximal block motifs found.

Theorem 30. ALGORITHM LM *correctly identifies all longest block motifs in a string X, when the matching rule for the central part is given by a partition table. It can be implemented to run in $O(n \log n)$ time.*

Proof. The proof of correctness comes from Lemma 29. Indeed, processing $\mathcal{B}$ bottom up means searching for longest block motifs of the form $V \diamond^k W$, with the length of V decreasing along a path. At an internal node, we must check for new maximal block motifs, but by Lemma 29, the number of new candidates is limited. Indeed, for each element of $\mathcal{A}_1$, we need to check only two elements of $\mathcal{A}_2$, via LCA queries.

In turn, such a selection of candidates can be efficiently implemented by a careful use of data structures for merging and a processing of "merging" operations according to the "smaller half trick". Indeed, at the beginning of the algorithm, each label at a leaf is in a list by itself. At each internal node, the list of its children is merged, being careful to merge the smaller into the bigger one. Finally, the lists are represented as balanced search trees, and an optimal merging procedure is used. The details of the analysis are as in [8]. Finally, we need to build both $T_{\overline{X}}$ and T'_X, which can be done in $O(n \log n)$ time ([3, 18] and Lemma 25). □

5 Longest Block Motifs with Either the Don't Care Table or with Renaming Functions

In this Section we show how to specialize the algorithm in Section 4 when the central part is specified either by $\mathcal{T}_\diamond$ or $\mathcal{T}_r$. All we need to do is to define two lexicographic order relations, analogous to the one in Definition 24. In turn, that will enable us to define two new versions of variants of the tree T'_X. Each of the two can still be built in $O(n \log n)$ time with `Algorithm BuildTree` and used in `Algorithm LM` to identify each type of motifs. We limit ourselves to defining these new trees. For the new objects we define, we keep the same notation as for their analogous in Section 4.

5.1 The Don't Care Table

Let $*$ be a symbol not belonging to the alphabet and not matching any other symbol of the alphabet. Consider Definition 22 and change it as follows:

Definition 31. *The modified prev representation of a string Y, denoted as mprev(Y), is defined as follows. If $|Y| = m \leq k$, then it is $*^m$. Else, it is $*^k$prev(Z), where $Y = WZ$ and $|W| = k$. For instance, let $Y = abauuxx$, and $k = 3$. Then, its modified prev representation is mprev$(Y) = ***0101$.*

We now define another Patricia Tree, still denoted by T'_X. Consider Definition 23 and change it as follows:

Definition 32. *Let X be a string with a unique endmarker. Let T'_X be a Patricia Tree storing each suffix of X, via their mprev representation according to Definition 31. That is, T'_X is like a p-suffix tree, but the initial part of each suffix is represented with $*$'s. For instance, let $X = abbabbb$ and $k = 2$, the first suffix of X is stored as $**babbb$.*

Finally, consider Definition 24 and change it as follows:

Definition 33. *Let Y and Z be two strings. We define a new order relation $Y \leq_m Z$ as follows. When $|Y| \leq k$, it must be $|Y| \leq |Z|$. Assume that $|Y| > k$, and let $Z = US$ and $Y = RP$, with $|R| = |U| = k$. Then, it must be $prev(P) \leq_2 prev(S)$. With a little abuse of notation, we can write $mprev(Y) \leq_m mprev(Z)$.*

Observe that `Algorithm BuildTree` will work correctly with this new definition of lexicographic order, except that now, in order to compare suffixes, we need only the p-suffix tree $T_{\hat{X}}$. Finally, the results in Section 4.2 hold verbatim:

Theorem 34. `ALGORITHM LM` *correctly identifies all longest block motifs in a string X, when the matching rule for the central part is given by the don't care table. It can be implemented to run in $O(n \log n)$ time.*

5.2 Renaming Functions

Consider Definition 22 and change it as follows:

Definition 35. *The modified prev representation of a string Y, denoted by mprev(Y), is defined as follows. If $|Y| \leq k$, then it is prev(Y). Else, it is prev(W)prev(Z), where $Y = WZ$ and $|W| = k$. For instance, let $Y = xuxuuxx$, and $k = 3$. Then, its modified prev representation is mprev$(Y) = 0020101$. Notice that its prev representation is 0022131.*

We now define another Patricia Tree, still denoted by T'_X. Consider Definition 23 and change it as follows:

Definition 36. *Let X be a string with a unique endmarker symbol. Let T'_X be a Patricia Tree storing each suffix of X, via their mprev representation according to Definition 35.*

Finally, consider Definition 24 and change it as follows:

Definition 37. *Let Y and Z be two strings. We define a new order relation $Y \leq_m Z$ as follows. When $|Y| \leq k$, it must be prev$(Y) \leq_2$ prev(Z). Assume that $|Y| > k$, and let $Z = US$ and $Y = RP$, with $|R| = |U| = k$. Then, it must be $prev(R) <_2 prev(U)$ or $prev(R) = prev(U)$ and $prev(P) \leq_2 prev(S)$. With a little abuse of notation, we can write $mprev(Y) \leq_m mprev(Z)$.*

Observe that `Algorithm BuildTree` will work correctly with this new definition of lexicographic order, except that now we again need only the p-suffix tree $T_{\hat{X}}$ in order to compare two suffixes in X. Finally, the results in Section 4.2 hold verbatim:

Theorem 38. `ALGORITHM LM` *correctly identifies all longest block motifs in a string X, when the matching rule for the central part is given by renaming functions. It can be implemented to run in $O(n \log n)$ time.*

6 Further Extensions

In the previous Sections, we have considered the notion of block motif with respect to tables and functions. In particular, the central part is constrained by a notion of match given by "rules" described by "functions". However, as it should be clear from the presentation in Section 4, the entire approach proposed here will work as long as we can define a linear order relation on strings analogous to Definition 24. We outline here a case in which that can be done and that cannot be expressed by "functions". We limit ourselves to consider standard strings.

Definition 39. *Let $i_1, j_1, \cdots, i_s, j_s$ be a given sequence of integers such that their sum is equal to k. Given two strings C and D we say that they match if and only if there exists a regular expression $*^{i_1} Z_1 *^{i_2} Z_2 \cdots *^{i_s} Z_s$, generating both strings, where $*$ denotes a don't care character and Z_f is a string of length j_f. That is, apart from the don't care regions, the two strings are the same*

For convenience of the reader, we now state the definition of a general k repeat, and of a corresponding motif when the matching rule is given by the regular expression in Definition 39. They are the analogous to Definitions 9 and 11.

Definition 40. Let Y be a factor of X. Y is a *general k-repeat*, with respect to the regular expression in Definition 39, if and only if the following conditions hold: (a) Y can be written as VQW, V and W both non-empty and $|Q| = k$ (b) there exists another factor $Z = V'Q'W'$ of X, such that $V = V'$, $W = W'$ and Q and Q' are both generated by the regular expression in Definition 39.

Definition 41. *Let $R(k, i, j)$ be the following binary relation on strings of length m, with $j < m$ and $k = j - i + 1$: $ZR(k, i, j)Y$ if and only if $(z_1 z_2 \cdots z_{i-1})$, $(z_{j+1} \cdots z_m)$ and $(y_1 x_2 \cdots y_{i-1})$, $(y_{j+1} \cdots y_m)$ exact match, respectively, while $(z_i \cdots z_j)$ and $(y_i \cdots y_j)$ are generated by the regular expression in Definition 39, $1 < i \leq j < m$.*

Since R is an equivalence relation, we can define the corresponding motifs:

Definition 42. *Given a string X, consider a factor Y, of length m, and assume that it is a k-repeat. Fix i and j as in Definition 41 and consider all factors Z of X such that $YR(k,i,j)Z$.Then, for each class with at least two elements, a block motif is any arbitrarily chosen factor in that class, say Y. As for standard strings, the block motif can be written as $y_1y_2\cdots y_{i-1}\diamond^k y_{j+1}\cdots y_m$, once it is understood that $\diamond^k$ is a place holder specifying a central part of the motif and that the matching criterion for that part is given by the regular expression in Definition 39.*

Consider Definition 22 and change it as follows:

Definition 43. *Let Y be a string of length m. Let f be the maximum index such that $\Sigma_{h=1}^{f}(i_h + j_h) \leq m$. Then prev$(Y) = *^{i_1}Y_1 *^{i_2} Y_2\cdots *^{i_f} Y_f$, where Y_h is the factor of Y following the don't care part $*^{i_h}$.*

Again, we can define another Patricia Tree, still denoted by T'_X. Consider Definition 23 and change it as follows:

Definition 44. *Let X be a string with a unique endmarker symbol. Let T'_X be a Patricia Tree storing each suffix of X, via their mprev representation according to Definition 43.*

Now define a lexicographic order relation on strings induced by their *prev* representation. That is, X is lexicographically smaller than Y if and only if *prev*$(X) \leq$*prev*(Y). Observe that `Algorithm BuildTree` will work correctly with this new definition of lexicographic order, except that now we need only the suffix tree $T_{\hat{X}}$ in order to compare two suffixes in X. Moreover, such a comparison can be made in $O(s)$ time, yielding a total of $O(sn\log n)$ time. Since $s \leq k$ is fixed and independent of n, we have once again that the results in Section 4.2 hold verbatim:

Theorem 45. `ALGORITHM LM` *correctly identifies all longest block motifs in a string X, when the matching rule for the central part is given by the regular expression in Definition 39. It can be implemented to run in $O(n\log n)$ time.*

7 Conclusions

We have considered a new notion of motif. It is of the form $V\diamond^k W$, where k is fixed. The central part of the motif can be more constrained than what is usually allowed by the don't care regular expression. We have also investigated several matching criteria for the central part, all related to parameterized strings and functions. Standard strings are a special case. Moreover, we have also shown that our framework can be extended to "central parts" given by regular expressions.

We have also presented a single algorithm that can be specialized to deal with all of the cases discussed here, showing that the *the two tree trick* devised by Crochemore et al. [8] can be extended to yield a general technique to identify the loongest motifs of the kind presented in this paper.

References

1. A. Amir, Y. Aumann, R. Cole, M. Lewenstein, and Ely Porat. Function matching: Algorithms, applications, and a lower bound. In *Proc. of ICALP 03, Lecture Notes in Computer Science*, pages 929–942, 2003.
2. B. S. Baker. Parameterized pattern matching: Algorithms and applications. *J. Comput. Syst. Sci.*, 52(1):28–42, February 1996.
3. B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Computing*, 26(5):1343–1362, October 1997.
4. A. Brazma, I. Jonassen, I. Eidhammer, and D. Gilbert. Approaches to the automatic discovery of patterns in biosequences. *J. of Computational Biology*, 5:277–304, 1997.
5. G.S. Brodal, R.B. Lyngsø, C.N.S. Pederson, and J. Stoye. Finding maximal pairs with bounded gaps. *J. of Discrete Algorithms*, 1(1):1–27, 2000.
6. H.S. Chan and K.A. Dill. Compact polymers. *Macromolecules*, 22:4559–4573, 1989.
7. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms – Second Edition*. MIT Press, Cambridge, MA, 1998.
8. M. Crochemore, C. Iliopoulos, M. Mohamed, and M.F. Sagot. Longest repeats with a block of don't cares. In *Proc. of Latin 04, to appear, Lecture Notes in Computer Science*, 2004.

9. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
10. D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. on Computing*, 13:338–355, 1984.
11. S. Karlin and G. Ghandour. Multiple-alphabet amino acid sequence comparisons of the immunoglobulin kappa-chain constant domain. *Proc. Natl. Acad. Sci. USA*, 82(24):8597–8601, December 1985.
12. R. Kolpakov and G. Kucherov. Finding repeats with fixed gaps. In *Proc. of SPIRE 02,*, pages 162–168, 2002.
13. T. Li, K. Fan, J. Wang, and W. Wang. Reduction of protein sequence complexity by residue grouping. *Protein Eng.*, (5):323–330, 2003.
14. X. Liu, D. Liu, J. Qi, and W.M. Zheng. Simplified amino acid alphabets based on deviation of conditional probability from random background. *Phys. Rev E*, 66:1–9, 2002.
15. M. Lothaire. *Applied Combinatorics on Words*. in preparation.
16. M. Lothaire. *Combinatorics on Words*. Cambridge University Press, 1997.
17. M. Lothaire. *Algebraic Combinatorics on Words*. Cambridge University Press, 2002.
18. E.M. McCreight. A space economical suffix tree construction algorithm. *J. of ACM*, 23:262–272, 1976.
19. L.R. Murphy, A. Wallqvist, and R.M. Levy. Simplified amino acid alphabets for protein fold recognition and implications for folding. *Protein. Eng.*, 13:149–152, 2000.
20. B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *Siam J. on Computing*, 17:1253–1262, 1988.
21. M. Spitzer, G. Fuellen, P. Cullen, and S. Lorkowsk. Viscose: Visualisation and comparison of consensus sequences. *Bioinformatics, to appear*.
22. J. Wang and W. Wang. A computational approach to simplifying the protein folding alphabet. *Nat. Struct. Biol.*, 11:1033–1038, 1999.