Longest Motifs with a Functionally Equivalent Central Block

Maxime Crochemore[1,2,⋆], Raffaele Giancarlo[3,⋆⋆], and 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, UK
[3] Dipartimento di Matematica ed Applicazioni, Universitá di Palermo
Via Archirafi 34, 90123 Palermo, Italy
raffaele@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 symbol renaming, initially unknown), and a functionally equivalent central block. Such three-part motifs are called *longest block motifs*. 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 or 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 repeats with a block of k don't care symbols, where k is fixed. These are words

⋆ Partially supported by CNRS, France, the French Ministry of Research through ACI NIM, and by 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 and by the French Ministritry of Research through ACI NIM.

⋆⋆⋆ Partially supported by French Ministry of Research Programs BioInformatique Inter EPST and ACI NIM and by Wellcome Foundation, Royal Society and Nato Grants.

A. Apostolico and M. Melucci (Eds.): SPIRE 2004, LNCS 3246, pp. 298–309, 2004.

of the form $V \diamond^k W$ that appear repeated in a string X, where $\diamond^k$ is a region of length k with an arbitrary content. Their work has some relation with previous work on repeats with bounded gaps [5, 12]. In general, the term *motif* [9] is 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 words 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 [15–17]) 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 to the notion of parameterized strings and has given several algorithms that find maximal repeated words in a string that p-match, *i.e.*, that are identical up to a renaming (initially unknown) 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 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 words 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 this paper, we make a first 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 motifs, *i.e.*, motifs 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 $\diamond^k$, which usually indicates a don't care block of length k, assumes in the case of the present paper a new meaning. Indeed, $\diamond^k$ is now 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 block*, 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 block identical, up to a renaming of symbols. For instance, *abxyab* and *ababab* are both described by $ab \diamond^2 ab$ and the given rule, since there is a one-to-one correspondence between $\{x, y\}$ and $\{a, b\}$. Notions associated with the example and the intuition just given are formalized in Section 3, where the central block $\diamond^k$ is specified by a set of matching criteria, all related to parameterized strings and function matching. Moreover, our approach can be extended to the case where such central block is a fixed regular expression, containing no Kleene closure or complement operation. Our main contribution for this part is a formal treatment of this extended type of motifs, resulting in conditions under which their definition is sound.

At the algorithmic level, our main contribution is to provide a general algorithm that extracts all longest block motifs, occurring 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 Section 3 the general algorithm specializes to find that type of motif by simply defining a new lexicographic order relation on strings. We also 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 even subtler repeated patterns in a string.

Due to space limitations, proofs will either be omitted or simply outlined. Moreover, we shall discuss only some of the block motifs that can be identified by our algorithm.

2 Preliminaries

2.1 Parameterized Strings

We start by recalling some basic definitions from the work by Brenda Baker on parameterized strings [2, 3]. Let Σ and Π be two alphabets, referred to as *constant* and *parameter*, 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 of equal length p-match if and only if there exists a bijective morphism $G : \Sigma \cup \Pi \rightarrow \Sigma \cup \Pi$ such that $G(\alpha) = \alpha$ for $\alpha \in \Sigma$ and $Y_i = G(X_i)$, $\forall i \in [1..|X|]$.*

For instance, $X = abuvabuvu$ and $Y = abxyabxyx$ p-match, with G such that $G(u) = x$ 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 bijective morphims over Σ_1 as *renaming functions*. We also use the term prefix, suffix and word 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 need to recall the definition of parameterized suffix tree, denoted by p-suffix tree, also due to Baker [2, 3]. Its definition is based, among other things, on a transformation of suffixes and prefixes of a string such that, when they match, they can share a path in a lexicographic tree. Indeed, consider the string $Y = uuuvvv$, made only of the parameters u and v. Notice that uuu and vvv p-match, and therefore they should share a path when the suffixes of the string are "stored" in a (compacted or not) lexicographic tree. That would not be possible if the lexicographic tree 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.

Let *prev* be a *transformation function* on strings operating as follows on a string X. For each parameter, its first occurrence in X is replaced by 0, and each successive occurrence is represented by its distance, along the string, to the previous occurrence. Constants are left unchanged. We denote by $prev(X)$ the *prev representation* of X over the alphabet Σ_2.

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]:

Lemma 2. *Two strings X and Y p-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 words, in each string, are a p-match. For instance, consider *abxyxzaaya* and *zzzztzwaata*. Words *xyxzaaya* and *ztzwaata* match, but that cannot be directly inferred from the *prev* representation of the two full strings.

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 lexicographic tree storing the *prev* representation of all suffixes of X.

The above definition is sound in the sense that all factors of X are represented in the p-suffix tree (that follows from the fact that each such word 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 2). Therefore, the two words 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 the prev representation of strings. 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 3. *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 $\leq_2$.*

2.2 Matching via Functions

In what follows, we need another type of relation that, for now, we define as a *Table*. A *Table* T has domain Σ_1 and ranges over the power set of Σ_1.

Definition 4. *Given two tables T and T' and two strings X and Y of length n, we say that X table matches Y via the two tables T and T', or, for short, that X and Y t-match, if and only if $y_i \in T(x_i)$ and $x_i \in T'(y_i)$, for all $1 \leq i \leq n$.*

For instance, 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 first difference between table and parameterized matches is that in the first case, all symbols in Σ_1 are treated as parameters and the correspondence is fixed once and for all.

A more substantial difference between table and parameterized matches is that tables may not be functions (as in the example above). For arbitrary tables, t-matching is also in general not an equivalence relation. Indeed, although symmetry is implied by the definition, neither reflexivity nor transitivity are. Notice also that t-matching incorporates the notion of match with don't care. In this latter case, both tables assign to each symbol the don't care symbol. We call this table the *don't care* table.

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 parameterized and table match.

Let $\mathcal{T}$ be a family of tables and k an integer, with $0 \leq k \leq n$, where n is the length of a string X. Consider also a family of renaming functions.

Definition 5. *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 word Z of X, two renaming functions F and G and two tables in $\mathcal{T}$, such that $Z = F(V)Q'G(W)$ and Q and Q' t-match, via the two tables.*

Definition 6. *Let $R(k, i, j)$ be the following binary relation on strings of length m, with $1 < i \leq j + 1$, $j < m$ and $k = j - i + 1$: Z $R(k, i, j)$ Y if and only if $(z_1 z_2 \cdots z_{i-1})$, $(z_{j+1} \cdots z_m)$ and $(y_1 y_2 \cdots y_{i-1})$, $(y_{j+1} \cdots y_m)$ match, respectively, while $(z_i \cdots z_j)$ and $(y_i \cdots y_j)$ t-match via two not necessarily distinct tables in $\mathcal{T}$.*

We now give a formal definition of motif. Intuitively, it is a representative string that describes multiple occurrences of "equivalent" strings.

Definition 7. *Given a string X, consider a factor Y of X, of length m, and assume that it is a general k-repeat. Let i and j be as in Definition 6 and consider all factors Z of X such that Y $R(k, i, j)$ 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 word 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.*

For instance, restrict the family of tables to be the don't care table only. Let $Z = abvvva$ and $Y = abxxya$; then we have Z $R(2, 3, 4)$ 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 should be clear from the example discussed earlier, the notion of block motifs, as defined in [8], is a special case of the ones defined here. It is also clear that the family of all tables yields the same notion of block motif as the one with the don't care table only. However, it can be shown that exclusion of the don't care table is not enough to obtain a proper definition of block motifs. Fortunately, there are easily checkable sufficient conditions ensuring that the family of tables guarantees R to be an equivalence relation, as we outline next.

Definition 8. *Consider two tables T and T'. Let their composition, denoted by $\circ$, be defined by $T \circ T'(a) = \bigcup_{c \in T'(a)} T(c)$, for each symbol a in the alphabet. Tha family $\mathcal{T}$ is closed under composition if and only if, for any two tables in the family, their composition is a table in the family.*

Definition 9. *A table T contains a table T' if and only if $T'(a) \subseteq T(a)$, for each symbol a in the alphabet.*

Lemma 10. *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.*

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. Let us partition the alphabet into classes and let $\mathcal{P}$ denote the corresponding partition. We then 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. Such situations arise in practice (see for instance [6, 11, 13, 14, 19, 21, 22]), in particular in the study of protein folding.

Let the fourth family of tables consist of only $\mathcal{T}_\mathcal{P}$, for some given partition $\mathcal{P}$ of the alphabet Σ_1.

Lemma 11. *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 6 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 $\mathcal{T}_r$. Therefore, for all those tables one can properly define block motifs.*

Let the family of tables be one-to-one functions. Consider $X = YZ$, where $Y = abxxya$ and $Z = abvvva$. 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.

Let the family of tables be $T_{a,b}$, defined earlier. Let $Z = cdccdacdc$ and $Y = cdccdbcdc$. Let $X = ZY$. Then $cdc \diamond^3 cdc$ 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 one introduced in [8]. In fact, we show that the main techniques used there, and that we nickname as *the two-tree trick*, represent a 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 from 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 briefly outlined in section 5.

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 (parameterize) match in the positions corresponding to V and W. We are interested in finding all longest block motifs.

Consider a lexicographic 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 12. *The modified prev representation of a string Y, $mprev(Y)$, 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 13. *Let X be a string with a unique endmarker. Let T'_X be a lexicographic tree storing each suffix of X in lexicographic order, via its 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. We anticipate that we only need to build and use the topology of T'_X, since we do not use it for pattern matching and indexing, as it is costumary for those data structures.

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 14. *Let Y and Z be two strings. Let $\leq_3$ be a lexicographic order relation over Σ_3. 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 is a prefix of Z and $|U| = |Y|$. 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 $prev(P) \leq_2 prev(S)$. Abusing notation, we can write that $mprev(Y) \leq_m mprev(Z)$, when $Y \leq_m Z$.*

Let T be a tree and consider two nodes u and v. Let $LCA(u, v)$ denote the lowest common ancestor of u and v. Given the suffix tree $T_{\hat{X}}$ [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 positions. 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 lexicographic trees.

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 15. *Tree T'_X can be correctly built 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 shall 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 16. *We say that $V \diamond^k W$ is maximal if and only if extending any word 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 $V \diamond^k W$.*

For instance, let $X = aabbaxxbxababyyayabbbuu$. Block motif $ab\diamond^2 xx$ is maximal. Indeed, it represents the class of words $\{abbaxx, ababyy, abbbuu\}$. However, extending any of those words either to the right and to the left results in a smaller class.

Lemma 17. *Consider a string X, its reverse $\overline{X}$, 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.*

We also need the following:

Lemma 18. *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.*

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 18 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 19. 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 18. The details of the analysis are as in [8] with the addition that 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 15). □

5 Extensions

In this Section we show how to specialize the algorithm in Section 4 when the central part is specified by $\mathcal{T}_\diamond$. All we need to do is to define a lexicographic order relation, analogous to the one in Definition 14. In turn, that will enable us to define a variant of the tree T'_X, which can still be built in $O(n \log n)$ time with `Algorithm BuildTree` and used in `Algorithm LM` to identify block motifs with the don't care symbol. We limit ourselves to define the new tree. An analogous reasoning will yield algorithms dealing with a central part defined by either renaming functions or by regular expressions with no Kleene Closure or Complement operation. The details are omitted. For the new objects we define, we keep the same notation as for their analogous in Section 4.

Let $*$ be a symbol not belonging to the alphabet and not matching any other symbol of the alphabet. Consider Definition 12 and change it as follows:

Definition 20. *The modified prev representation of a string Y, 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 lexicographic tree, still denoted by T'_X. Consider Definition 13 and change it as follows:

Definition 21. *Let X be a string with a unique endmarker. Let T'_X be a lexicographic tree storing each suffix of X, via their mprev representation according to Definition 20. 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 14 and change it as follows:

Definition 22. *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 23. `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.*

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. Maxime Crochemore, Costas S. Iliopoulos, Manal Mohamed, and Marie-France Sagot. Longest repeated motif with a block of don't cares. In M. Farach-Colton, editor, *Latin American Theoretical INformatics (LATIN)*, number 2976 in LNCS, pages 271–278. Springer-Verlag, 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. *Combinatorics on Words*. Cambridge University Press, 1997.
16. M. Lothaire. *Algebraic Combinatorics on Words*. Cambridge University Press, 2002.
17. M. Lothaire. *Applied Combinatorics on Words*. in preparation, 2004. http://igm.univ-mlv.fr/∼berstel/Lothaire/index.html.
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*, 2004.
22. J. Wang and W. Wang. A computational approach to simplifying the protein folding alphabet. *Nat. Struct. Biol.*, 11:1033–1038, 1999.