Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 362 (2006) 248–254
Theoretical Computer Science
www.elsevier.com/locate/tcs

Longest repeats with a block of k don't cares

Maxime Crochemore[a,b,1], Costas S. Iliopoulos[b,2], Manal Mohamed[b,*], Marie-France Sagot[c,3]

[a] *Institut Gaspard-Monge, University of Marne-la-Vallée, 77454 Marne-la-Vallée CEDEX 2, France*
[b] *Department of Computer Science, King's College London, London WC2R 2LS, UK*
[c] *Inria Rhône-Alpes, Laboratoire de Biométrie et Biologie Évolutive, Université Claude Bernard, 69622 Villeurbanne cedex, France*

Received 11 March 2005; received in revised form 5 June 2006; accepted 27 June 2006

Communicated by Aviezri Fraenkel

Abstract

A *k-repeat* is a string $w_k = u *^k v$ that matches more than one substring of x, where $*$ is the don't care letter and $k > 0$. We propose an $O(n \log n)$-time algorithm for computing all longest k-repeats in a given string $x = x[1..n]$. The proposed algorithm uses suffix trees to fulfill this task and relies on the ability to answer lowest common ancestor queries in constant time.

Keywords: Combinatorial pattern matching; String; Repeat; Don't care

1. Introduction

In biology the term *motif* is often used to describe similar functional components that different biological sequences (DNA, RNA, proteins) may have in common. For instance, in DNA sequences the sets could represent noncoding repeat elements, promoter sequences, regulatory sites or enhancers/silencers. In many cases, such as promoter sequences or regulatory sites, the motifs are *composite*, i.e., the functional components represented by such motifs are in fact composed of two or more strictly/approximaly conserved parts separated by random regions of variable lengths.

A *repeat* in a string x is a substring w of x that occurs more than once. For given fixed $k > 0$, a *k-repeat* is a string $w_k = u *^k v$ that matches more than one substring of x, where $*$ is the don't care letter. A *longest k-repeat* is a k-repeat of maximum length over x.

In this paper, we concentrate on calculating a special kind of motifs. By considering the biological sequence as a string, a k-repeat can be seen as a motif with two solid parts separated by a random region (*gap*) of length k. We propose an $O(n \log n)$-time algorithm for computing all longest k-repeats as well as their corresponding matching substrings in a given string of length n.

* Corresponding author. Tel.: +44 207 8482492; fax: +44 207 8482851.
E-mail addresses: maxime.crochemore@univ-mlv.fr (M. Crochemore), csil@dcs.kcl.ac.uk (C.S. Iliopoulos), manal@dcs.kcl.ac.uk (M. Mohamed), Marie-France.Sagot@inria.fr (M. Sagot).

[1] Partially supported by CNRS, Wellcome Foundation and NATO grants.
[2] Partially supported by a Marie Curie fellowship, Wellcome Foundation, NATO and Royal Society grants.
[3] Partially supported by French Programme BioInformatique Inter EPST, Wellcome Foundation, Royal Society and NATO grants.

0304-3975/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2006.06.029

The paper is organised as follows: In Section 2, we state the preliminaries used throughout the paper. In Section 3, we define the k-repeats problem and describe in general how to compute all longest k-repeats using two suffix trees. In Section 4, we present our method to speed up the computation. In Section 5, we give details of the algorithm. In Section 6, we analyse the running time of the algorithm. Conclusions are drawn in Section 7.

2. Preliminaries

Throughout the paper, a *string* x of length n is considered as $x[1..n] = x[1]x[2]\cdots x[n]$, where $x[i] \in \Sigma$ is the *ith letter* of x and Σ is a finite alphabet. The number of letters in x is called the *length* of x, and is denoted by $|x|$. The *empty* string (of length zero) is denoted by ε. The *reverse* of x is denoted by $\overleftarrow{x}$.

A symbol '$*$' $\notin \Sigma$ is called a *don't care* letter. The don't care letter *matches* any letter in the alphabet. Two letters λ and μ are said to *match* ($\lambda = \mu$) if they are equal or one of them is don't care.

A *repeat* in a string x is a substring w of x that occurs more than once. Let $\mathcal{N}_w$ be the number of times w occurs in x. That is, if $L_w = \{i_1, i_2, ...\}$ is the complete *occurrences-list* of w in x, then $\mathcal{N}_w = |L_w|$. A repeat w is called *left-maximal* if and only if $\mathcal{N}_{\lambda w} \neq \mathcal{N}_w$ for any $\lambda \in \Sigma - \{\varepsilon\}$. Respectively, a repeat w is called *right-maximal* if and only if $\mathcal{N}_{w\mu} \neq \mathcal{N}_w$ for any $\mu \in \Sigma - \{\varepsilon\}$. If w is both left-maximal and right-maximal then it is said to be *maximal*.

For given $k > 0$, a *k-repeat* is a string $w_k = u *^k v$ that matches more than one substring of x, where $*$ is the don't care letter, and u and v are both over Σ. We will call u and v the *left-part* and *right-part* of the w_k, respectively. The *longest k-repeat* is a k-repeat of maximum length over x. Note that the longest k-repeat in x in not necessarily unique. Here, ℓ_k^* is used to denote the length of the longest k-repeat in a given string x. Note that, if ℓ^* is the length of the longest repeat in x then $\ell_k^* \geqslant \ell^* + k$.

We present a method for finding all longest k-repeats in a given string x. Our method uses the suffix tree of x as a fundamental data structure. A complete description of suffix trees is beyond the scope of this paper, and can be found in [7,8]. However, for the sake of completeness, we will briefly review the notion.

Definition 1 (*Suffix tree*). A suffix tree $\mathcal{T}(x)$ of the string $x\$ = x[1..n]\$$, $n \geqslant 1$ is a rooted directed tree with exactly n leaves numbered 1 to $n+1$, where $\$ \notin \Sigma$. Each internal node has at least two children and each edge is labelled with a nonempty substring of x. No two edges descending from a node can have edge-labels beginning with the same letter. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the ith suffix $x[i..n+1]$ of x, with $n+1$ denoting the empty suffix.

For any node v, the *path-label* of v is the label of the path from the root of $\mathcal{T}(x)$ to v; it is denoted by $label(v)$. The *string-depth* of v is the number of letters in v's path-label; it is denoted by $depth(v)$. The *leaf-list* of v is the set of the leaf numbers in the subtree rooted at v; it is denoted by $LL(v)$.

Several algorithms construct the suffix tree $\mathcal{T}(x)$ in $\Theta(n)$ time and space, assuming fixed alphabet (see for example [7,8]). All the internal nodes in $\mathcal{T}(x)$ have an out-degree between 2 and $|\Sigma|$. Therefore, we can transform the suffix tree $\mathcal{T}(x)$ into a *binary* one $\mathcal{B}(x)$ by replacing every node v in $\mathcal{T}(x)$ with out-degree $d > 2$ by a binary tree with $d-1$ internal nodes and $d-2$ internal edges, where the d leaves are the d children of v. Since $\mathcal{T}(x)$ has n leaves, constructing $\mathcal{B}(x)$ requires adding at most $n-2$ new nodes. Each new node can be added in constant time. This implies that the binary suffix tree $\mathcal{B}(x)$ can be constructed in $O(n)$ time.

Our method makes use of Schieber and Vishkin's *lowest common ancestor* (*LCA*) algorithm [11]. For a given rooted tree $\mathcal{T}$, the LCA of two nodes u and v is the deepest node in $\mathcal{T}$ that is ancestor of both u and v. After a linear amount of preprocessing of a rooted tree, any two nodes can be specified and their lowest common ancestor found in constant time. That is, a rooted tree with n nodes is first preprocessed in $O(n)$ time, and thereafter any lowest common ancestor query takes only a constant time to be solved, independent of n.

In the context of suffix trees, the situation commonly arises that both u and v are leaves in $\mathcal{T}(x)$, where $x[i..n]$ and $x[j..n]$ are the suffixes represented by u and v, respectively, for integers i and j in the range $1..n+1$. In this case, the node $w = LCA(u, v)$ is the root of the minimum size subtree which contains u and v. Note that the path-label of w ($label(w)$) is the *longest common prefix* of $x[i..n]$ and $x[j..n]$. The ability to find a longest common prefix is an important primitive in many string problems.

3. The longest *k*-repeats problem

The *longest k-repeats* problem requires finding all longest k-repeats in a given string x together with their length ℓ_k^*. For example, $x = abcdabcaefabcgabc$ has 2-repeats

$$*^2,\ a*^2,\ *^2a,\ a*^2a,\ ab*^2aabc*^2d,\ ab*^2abc,\ abc*^2bc,$$

among others, of which the last two are the longest ($\ell_2^* = 7$). The first longest 2-repeat, $ab*^2abc$, matches both *abcdab* and *abcgabc*. The latter, $abc*^2bc$, matches both *abcdabc* and *abcgabc*.

By definition, a k-repeat matches at least two substrings in x. Let $w_k^* = u*^k v$ be one of the longest k-repeat in x that matches both $x[i..i+\ell_k^*]$ and $x[j..j+\ell_k^*]$, where $1 \leqslant i,\ j \leqslant (n-\ell_k^*)$. Therefore, the left-part u occurs at positions i and j and the right-part v occurs at positions $i+|u|+k$ and $j+|u|+k$. Clearly, with respect to these two occurrences, u needs to be left-maximal repeat while v needs to be right-maximal repeat in x, that is, $x[i-1] \neq x[j-1]$ and $x[i+\ell_k^*] \neq x[j+\ell_k^*]$. Otherwise, w_k^* cannot be a longest k-repeat.

The suffix tree provides a compact representation of all maximal repeats. In fact, all right-maximal repeats in x are represented naturally by the suffix tree of x. Similarly, all left-maximal repeats of x are represented by the suffix tree of $\overleftarrow{x}$. Thus, $\mathcal{T}(\overleftarrow{x})$ can be used to generate the left-part, while $\mathcal{T}(x)$ can be used to generate the right-part of the k-repeat. Note that the path-label ($label(v)$), of each internal node $v \in \mathcal{T}(\overleftarrow{x})$, represents the reverse of a left-maximal repeat in x ending at positions $\{j \mid j = n+1-i \text{ and } i \in LL(v)\}$.

An obvious approach to solve the longest k-repeats problem is as follows:

1. generate all possible left-maximal repeats u, together with their occurrences list L_u;
2. for each pair of its occurrence positions i_1 and i_2, calculate the right-maximal repeat v, if any, that occurs at both $j_1 = i_1 + |u| + k$ and $j_2 = i_2 + |u| + k$;
3. calculate the length of the corresponding k-repeat $w_k = \overleftarrow{u}\,*^k v$;
4. report the longest k-repeats.

The above can be achieved as follows: First, replace each index i in $\mathcal{T}(\overleftarrow{x})$ by $n+1-i+(k+1)$. Thus, for each node $u \in \mathcal{T}(\overleftarrow{x})$, the modified leaf-list $\widetilde{LL}(u) = \{j \mid j = n+1-i+(k+1) \text{ and } i \in LL(u)\}$. Then, traverse $\mathcal{T}(\overleftarrow{x})$ from bottom up, visiting each node in the tree. In detail, walk from the leaves upward, visiting a node u only after

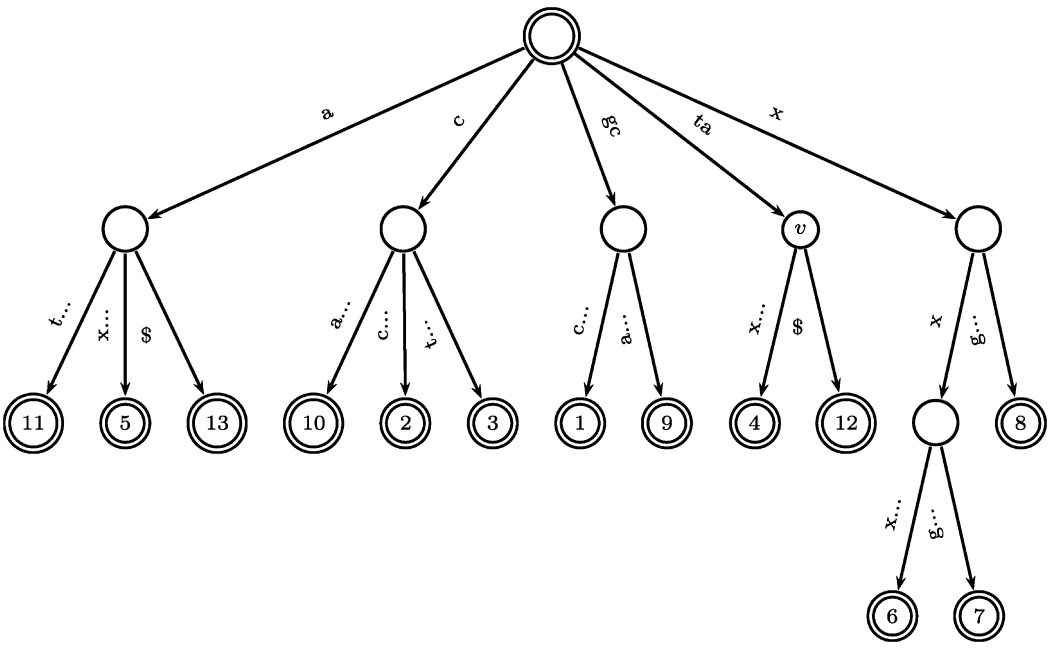

Fig. 1. The suffix tree of gcctaxxxgcata.

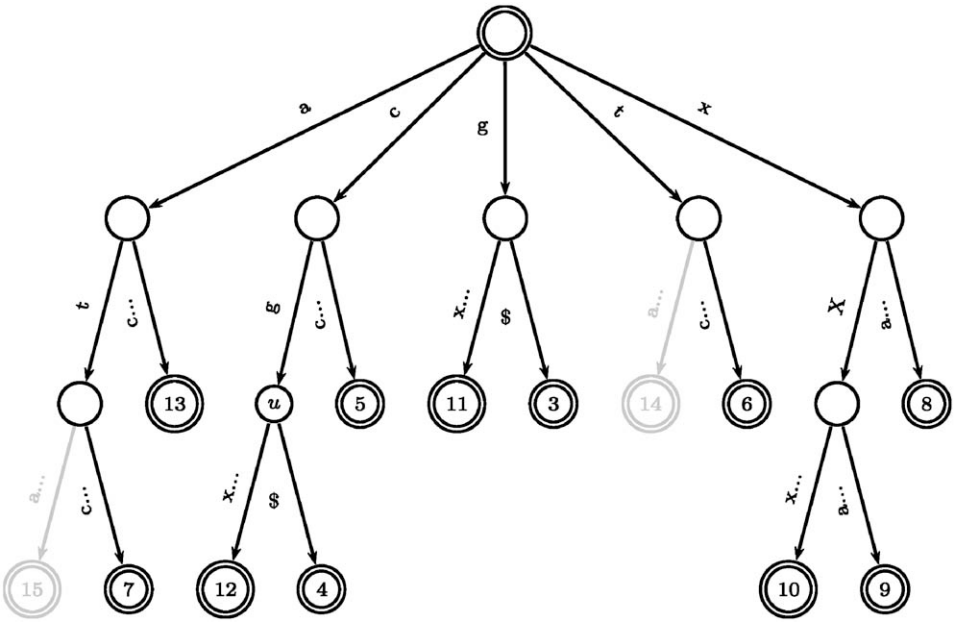

Fig. 2. The suffix tree of atacgxxxatccg. Each index i is replaced by $16 - i$. The gray nodes may be omitted.

visiting every child of u. During the visit to u, calculate for each possible pair $i_1, i_2 \in \widetilde{LL}(u)$ the node $v = LCA(i_1, i_2)$ in $\mathcal{T}(x)$. Finally, calculate the length of the k-repeat $w_k = \overleftarrow{label(u)} *^k label(v)$, which is equal to $depth(u) + k + depth(v)$.

For example, if $k = 1$ and $x =$ gcctaxxxgcata, then Figs. 1 and 2 represent the suffix trees of x and $\overleftarrow{x}$, respectively. Note that, each index i in $\mathcal{T}(\overleftarrow{x})$ has been replaced by $16 - i$. Consider node u in $\mathcal{T}(\overleftarrow{x})$ labelled by 'cg'. The modified leaf-list of u is $\widetilde{LL}(u) = \{4, 12\}$. Let node v be the lowest common ancestor of nodes 4 and 12 in $\mathcal{T}(x)$, $label(v) =$ 'ta'. Thus, $gc * ta$ is a 1-repeat. Since it is the longest 1-repeat in x, ℓ_1^* equals 5.

Clearly, this method takes $O(n^3)$ time. This is because there are $O(n)$ nodes in $\mathcal{T}(\overleftarrow{x})$ representing $O(n)$ possible candidates for the left-maximal left-part. For each node $u \in \mathcal{T}(\overleftarrow{x})$, the size of the modified leaf-list $\widetilde{LL}(u)$ is $O(n)$. Hence, there are $O(n^2)$ pairs to be checked for possible extension with a block of k don't cares and a right-maximal right-part. Checking each pair is done using a longest common ancestor query in $O(1)$ time. This gives a total of $O(n^3)$ time for this method. In the next section, we will show how this method can be modified to run in $O(n \log n)$ time.

4. Speeding up the calculation

In this section, we will explain how the method suggested in the previous section can be modified efficiently in order to avoid checking all pairs in the modified leaf-lists. The idea is based on two main observations. Let $\mathcal{B}(\overleftarrow{x})$ be the binary suffix tree of $\overleftarrow{x}$ and let u_1 and u_2 be the two children of node u. Moreover, let $\widetilde{LL}(u_1) = \{i_1, i_2, ...\}$ and $\widetilde{LL}(u_2) = \{j_1, j_2, ...\}$ be the modified leaf-lists of u_1 and u_2. Our first observation states that, if we traverse $\mathcal{B}(\overleftarrow{x})$ bottom-up, then at each internal node u and for all possible integers p and q, we only need to check all pairs i_p, j_q for possible extension with k don't cares and right-maximal right-part. Thus, there is no need to check pairs i_p, i_q. This is because such pairs have been already checked during the visit to some nodes v_1, where v_1 is in the subtree rooted at u_1. Similarly, we do not need to check pairs j_p, j_q. This is because such pairs have been also checked during the visit to some nodes v_2, where v_2 is in the subtree rooted at u_2.

The second observation is based on our interest in calculating all longest k-repeats and not in calculating all possible k-repeats. Recall that checking each pairs of indices for a possible extension is done using a constant-time longest common ancestor query. One of the most interesting properties of the lowest common ancestor is 'the leaves that are close together have a least common ancestor of greater depth'. In the context of suffix tree, the depth of a node is the length of the longest common prefix represented by that node. By traversing the suffix tree in a depth-first manner,

the suffixes of a string can be ordered by assigning a number to each leaf of the suffix tree; we will call this number a *preorder* number. Using the preorder numbers of the leaves, the previous property of the longest common ancestor can be extended to suffix trees as follows:

Lemma 1. *Let i, j and k be the preorder numbers given to three leaves u, v and w in a suffix tree $\mathcal{T}(x)$. If $i < j < k$, then the depth of $LCA(u, v)$ cannot be less than the depth of $LCA(u, w)$.*

Proof. Let s and t be $LCA(u, v)$ and $LCA(u, w)$, respectively. The proof is based on the following three facts:

1. Two nodes are in different subtrees of their least common ancestor.
2. For two subtrees of a node, all elements of the left subtree have a smaller preorder number than any element of the right subtree.
3. All ancestors of a node are on the path from it to the root.

Thus, if the depth of s is less than the depth of t then t must be on the path from u to s. Hence, t and any descendent of it must be in the same subtree of s as u, which is a subtree to the left of that of s containing v. This requires k to be less than j, which is a contradiction. □

This observation suggests the following: For each node $u \in \mathcal{B}(\overleftarrow{x})$, where u_1 and u_2 are the two children of u; and for all possible indices $i_p \in \widetilde{LL}(u_1)$, it is enough to check only two pairs i_p, j_q and i_p, j_r, where j_q and j_r are the closest leaves to leaf i_p in $\mathcal{T}(x)$ and $j_q, j_r \in \widetilde{LL}(u_2)$. Efficient calculation of j_q and j_r cannot be achieved without maintaining the indices in the modified leaf-lists in a sorted order according to their preorder numbering in $\mathcal{T}(x)$. In the next section, we present details of our algorithm and the data structure used.

5. Algorithm

The initialisation phase of the algorithm consists of two main steps. In the first step, the suffix tree $\mathcal{T}(x)$ is built and then traversed in a preorder manner where a number is assigned to each leaf. Let $no(i)$ be the preorder number assigned to the leaf i. For example, if $\mathcal{T}(x)$ is the tree of Fig. 1 then

i	1	2	3	4	5	6	7	8	9	10	11	12	13
$no(i)$	7	5	6	9	2	11	12	13	8	4	1	10	3

In the second step, the binary suffix tree $\mathcal{B}(\overleftarrow{x})$ is built. In addition, a list is associated with each leaf i, initialised with $no(n + 1 - i + (k + 1))$.

Note that the lists associated with the internal nodes of $\mathcal{B}(\overleftarrow{x})$ can be calculated during the bottom-up traversal of the binary suffix tree $\mathcal{B}(\overleftarrow{x})$. The list associated with each internal node u is the sorted union of the disjoint lists of the two children of u. This list can be considered as a modified leaf-list sorted according to the preorder numbers from $\mathcal{T}(x)$. In order to guarantee an efficient merge of the lists, they are implemented using AVL-trees [3]. Although this implementation is similar to the one used in [4,9], any other type of balanced search trees may be used. The efficient merging of two AVL trees is essential to our method. The results on the merge operations of two height-balanced trees stated in [5] are summarised in the following lemmas.

Lemma 2. *Two AVL trees of size at most n and m can be merged in time* $\mathrm{O}\left(\log \binom{n+m}{n}\right)$.

Lemma 3. *Given a sorted list of elements $e_1 \leqslant e_2 \leqslant \cdots \leqslant e_n$, and an AVL tree T of size at most m, where $m \geqslant n$, we can find $q_i = \max\{x \in T \mid x \leqslant e_i\}$ for all $i = 1, 2, \ldots, n$ in time* $\mathrm{O}(\log \binom{n+m}{n})$.

Using the *smaller-half trick*, which states that '*the sum over all nodes v of an arbitrary binary tree of terms that are* $\mathrm{O}(n_1)$, *where n_1 and n_2 are the numbers of leaves in the subtrees rooted at the children of v and $n_1 \leqslant n_2$, is* $\mathrm{O}(n \log n)$', the following lemma stated in [4] is easy to prove:

Lemma 4. *Let T be an arbitrary binary tree with n leaves. The sum over all internal nodes v in T of terms* $\log \binom{n_1+n_2}{n_1}$, *where n_1 and n_2 are the numbers of leaves in the subtrees rooted at the two children of v, is* $\mathrm{O}(n \log n)$.

Algorithm *All-Longest-k-Repeats*(x, k)
Input: A string x of length n and an integer k.
Output: The value ℓ_k^* and all longest k-repeats in x.
1. Initialisation:
2. Build the suffix tree $\mathcal{T}(x)$.
3. **for** each leaf $i \in \mathcal{T}(x)$ **do**
4. $no(i) \leftarrow$ the preorder number of i
5. Build the binary suffix tree $\mathcal{B}(\overleftarrow{x})$.
6. **for** each leaf $i \in \mathcal{B}(\overleftarrow{x})$ **do**
7. $\mathcal{A}_i \leftarrow \{no(n+1-i+(k+1))\}$
8. $max \leftarrow 0$
9. $M \leftarrow \emptyset$
10. Main Algorithm:
11. **for** each node $u \in \mathcal{B}(\overleftarrow{x})$ in a depth-first order **do**
12. $\mathcal{A}_{u_1}, \mathcal{A}_{u_2} \leftarrow$ the AVL trees of the two children of u, where $|\mathcal{A}_{u_1}| \leq |\mathcal{A}_{u_2}|$
13. **for** all $p \in \mathcal{A}_{u_1}$ in ascending order **do**
14. $q \leftarrow \max\{j \in \mathcal{A}_{u_2} \mid j \leq p\}$
15. $v_{pq} \leftarrow LCA(no^{-1}(p), no^{-1}(q))$, where $no^{-1}(p), no^{-1}(q) \in \mathcal{T}(x)$
16. $r \leftarrow next(q, \mathcal{A}_{u_2})$
17. $v_{pr} \leftarrow LCA(no^{-1}(p), no^{-1}(r))$, where $no^{-1}(p), no^{-1}(r) \in \mathcal{T}(x)$
18. **if** $depth(v_{pq}) \geq depth(v_{pr})$
19. **then** $v \leftarrow v_{pq}$
20. **else** $v \leftarrow v_{pr}$
21. **if** $max = depth(u) + k + depth(v)$
22. **then** $M \leftarrow M \cup (u, v)$
23. **else** **if** $max < depth(u) + k + depth(v)$
24. **then** $max \leftarrow depth(u) + k + depth(v)$
25. $M \leftarrow (u, v)$
26. $\mathcal{A}_u \leftarrow merge(\mathcal{A}_{u_1}, \mathcal{A}_{u_2})$
27. **return** (max, M)

Fig. 3. *All-Longest-k-Repeats* algorithm.

The algorithm for calculating all longest k-repeats and their length ℓ_k^* in a given string x is presented in Fig. 3. Recall that at every node u in the binary suffix tree $\mathcal{B}(\overleftarrow{x})$, we construct an AVL tree $\mathcal{A}_u$ that stores the sorted list of all the preorder numbers associated with the elements in $\widetilde{LL}(u)$. If u is a leaf, then $\mathcal{A}_u$ is constructed directly (Line 7). If u is an internal node, then $\mathcal{A}_u$ is constructed by merging $\mathcal{A}_{u_1}$ and $\mathcal{A}_{u_2}$ (Line 24), where $\mathcal{A}_{u_1}$ and $\mathcal{A}_{u_2}$ are the AVL trees associated with the two children of u and $|\mathcal{A}_{u_1}| \leqslant |\mathcal{A}_{u_2}|$. Before constructing $\mathcal{A}_u$, we use $\mathcal{A}_{u_1}$ and $\mathcal{A}_{u_2}$ to check for an occurrence of longest k-repeat. If an integer p in $\mathcal{A}_{u_1}$ is going to be inserted between q and r in $\mathcal{A}_{u_2}$, then q and r are efficiently obtained (Lemma 3). Let v_p, v_q and v_r be the leaves in $\mathcal{T}(x)$ with a preorder number p, q and r, respectively. The algorithm calculates the string-depth of both $v_{pq} = LCA(v_p, v_q)$ and $v_{pr} = LCA(v_p, v_r)$. Let $v = v_{pq}$ if $depth(v_{pq}) \leqslant depth(v_{pr})$, v_{pr} otherwise. Then, the algorithm checks whether $depth(u) + k + depth(v)$ is greater than *max*, where *max* is the current calculated length of the longest k-repeat. If so, *max* is updated and the pairs (u, v) are stored in a list M. The algorithm returns the value ℓ_k^* and the list M. Each element in M is a pair of vertices (u, v) representing one of the longest k-repeats that is equal to $\overleftarrow{label(u)} *^k label(v)$, where $v \in \mathcal{T}(x)$ and $u \in \mathcal{B}(\overleftarrow{x})$. Since $u \in \mathcal{B}(\overleftarrow{x})$, where $\mathcal{B}(\overleftarrow{x})$ is obtained from the original suffix tree by adding some internal nodes, more checking is needed to avoid adding pairs to M representing the same k-repeat. Such checking is simple and should not affect the running time.

Generally, we are interested in reporting the list of occurrences of each longest k-repeat. This can be done by checking the leaf-lists of u and v or—perhaps more efficient—by running the algorithm twice once to calculate ℓ_k^* and again to report the occurrences list of each longest k-repeat. The latter will add a factor α to the original running time, where α is the output size.

6. Time complexity

In this section, we analyse the running time of the algorithm. Recall that, for constant size alphabet, a suffix tree can be built in linear time. Thus, building $\mathcal{T}(x)$ and giving preorder numbers for its leaves takes $O(n)$ time (Lines 2–4).

Similarly, building the binary suffix tree $\mathcal{B}(\overleftarrow{x})$ takes $O(n)$ time (Line 5). Creating an AVL tree of size one can be done in constant time. Therefore, doing so at each of the n leaves of $\mathcal{B}(\overleftarrow{x})$ takes $O(n)$ time (Lines 6–7).

The algorithm then traverses $\mathcal{B}(\overleftarrow{x})$ in depth-first manner (Lines 11–26). At every internal node u, the algorithm runs a search loop on Lines 13–25 and then performs a merge at Line 26. Let $\mathcal{A}_{u_1}$ and $\mathcal{A}_{u_2}$ be the two AVL trees associated with the two children of u where $|\mathcal{A}_{u_1}| \leqslant |\mathcal{A}_{u_2}|$. During the search loop, for each $p \in \mathcal{A}_{u_1}$, the algorithm searches $\mathcal{A}_{u_2}$ to find q and r. According to Lemma 3, the time required to complete the search loop at each node is $O(\log \binom{|\mathcal{A}_{u_1}|+|\mathcal{A}_{u_2}|}{|\mathcal{A}_{u_1}|})$. Additionally, Lemma 2 states that the merge at Line 27 takes also $O(\log \binom{|\mathcal{A}_{u_1}|+|\mathcal{A}_{u_2}|}{|\mathcal{A}_{u_1}|})$ time. Summing these terms over all the internal nodes of $\mathcal{B}(\overleftarrow{x})$ gives the total running time of the tree traversal, that is, $O(n \log n)$ (Lemma 4). Thus, the total running time of the algorithm is $O(n \log n)$ time. The following theorem states the result.

Theorem 1. *Algorithm All-Longest-k-Repeats calculates all longest k-repeats, in a given string x of length n, together with their length ℓ_k^* in $O(n \log n)$ time.*

7. Conclusions

We have presented an $O(n \log n)$ algorithm for computing all longest k-repeats in a given string $x = x[1..n]$. The algorithm uses two suffix trees intensively, one for the original string and the other for its reverse. The use of a generalised suffix tree (for both the string and its reverse) would be possible but is not necessary because we do not need all the information it contains. We have not yet explored the possibility of using an affix tree [12] but there are some doubt that it will lead to a significant improvement in the asymptotic time complexity.

The use of the suffix array is also a possible improvement. It was shown that the suffix array can be constructed and used as efficiently in terms of time [2] and more efficiently in terms of space than a suffix tree. In fact, the preorder array $no[1..n]$ introduced in Section 5 is somehow a suffix array; the exact suffix array can be achieved by ordering the leaves of the suffix tree lexicographically.

We remark also the similarity of our algorithm for the longest k-repeats and all-repeats algorithm [6] that also uses a reverse suffix tree, but works with suffix array as well. A more efficient all-repeats algorithm that uses suffix array and avoids the reverse suffix tree was presented in [1]. We left exploring the possibility of avoiding reverse suffix tree construction as an open problem.

Acknowledgements

The authors would like to express their gratitude to the referees of the Theoretical Computer Science for their knowledgeable and constructive comments that much improved the paper.

References

[1] M.I. Abouelhoda, S. Kurtz, E. Ohlebusch, The enhanced suffix array and its applications to genome analysis, in: Proc. Second Workshop on Algorithms in Bioinformatics, Lecture Notes in Computer Science, Vol. 2452, Springer, Berlin, 2002, pp. 449–463.
[2] M.I. Abouelhoda, S. Kurtz, E. Ohlebusch, Replacing suffix trees with enhanced suffix arrays, J. Discrete Algorithms 2 (2004) 53–86.
[3] G.M. Adel'son-Vel'skii, Y.M. Landis, An algorithm for the organization of information, Dokl. Akad. Nauk SSSR 146 (1962) 263–266.
[4] G.S. Brodal, R.B. Lyngsø, C.N.S. Pedersen, J. Stoye, Finding maximal pairs with bounded gap, J. Discrete Algorithms 1 (1) (2000) 77–104, (special issue of matching patterns).
[5] M.R. Brown, R.E. Tarjan, A fast merging algorithm, J. ACM 27 (2) (1979) 211–226.
[6] F. Franek, W.F. Smyth, Y. Tang, Computing all repeats using suffix arrays, J. Automata Languages and Combinatorics 8 (4) (2003) 579–591.
[7] M. Crochemore, W. Rytter, Jewels of Stringology, World Scientific, Singapore, 2002.
[8] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, Cambridge, 1997.
[9] C.S. Iliopoulos, C. Makris, S. Sioutas, A.K. Tsakalidis, K. Tsichlas, Identifying occurrences of maximal pairs in multiple strings, in: Proc. 13th Annu. Symp. on Combinatorial Pattern Matching—CPM'02, Springer, Berlin, Vol. 2373, 2002, pp. 133–143.
[11] B. Schieber, U. Vishkin, On finding lowest common ancestors: simplification and parallelization, SIAM J. Comput. 17 (6) (1988) 1253–1262.
[12] J. Stoye, Affix trees, Diploma Thesis, Technische Fakultät der Universität Bielefeld, Abteilung Informationstechnik, Report 2000–04, 2000.