HAL
open science

# Longest repeated motif with a block of don't cares

Maxime Crochemore, Costas S. Iliopoulos, Manal Mohamed, Marie-France Sagot

## To cite this version:

Maxime Crochemore, Costas S. Iliopoulos, Manal Mohamed, Marie-France Sagot. Longest repeated motif with a block of don't cares. 6th Latin American Theoretical Informatics (LATIN'04), 2004, Buenos Aires, Argentina. pp.271-278, 10.1007/978-3-540-24698-5_31 . hal-00619983

**HAL Id: hal-00619983**

**https://hal.science/hal-00619983v1**

Submitted on 18 Dec 2015

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Longest Repeats with a Block of Don't Cares

Maxime Crochemore[1,2⋆], Costas S. Iliopoulos[2⋆⋆], Manal Mohamed[2⋆⋆⋆], and Marie-France Sagot[3†]

[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
mac,csi,manal@dcs.kcl.ac.uk

[3] Inria Rhône-Alpes, Laboratoire de Biométrie et Biologie Évolutive,
Université Claude Bernard, 69622 Villeurbanne cedex, France
Marie-France.Sagot@inria.fr

**Abstract.** We introduce an algorithm for extracting all longest repeats with $k$ don't cares from a given sequence. Such repeats are composed of two parts separated by a block of $k$ don't care symbols. The algorithm uses suffix trees to fulfill this task and relies on the ability to answer the lowest common ancestor queries in constant time. It requires $O(n \log n)$ time in the worst-case.

**Keywords:** Combinatorial Problems, String, Repeat Extraction, Don't Care, Suffix Tree, Lowest Common Ancestor, Efficient Merging.

## 1 Introduction

In recent years, many combinatorial problems that originate in bioinformatics have been studied. Here we consider a combinatorial problem on motifs. The term *motif* [5] is often used in biology to describe similar functional components that several biological sequences have in common. It can also be used to describe any collection of similar substrings 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.

In this paper we explore a sub-problem that is important in the approach to the combinatorics and the complexity of the original biologically motivated topic. Thus, we concentrate on finding all *longest repeats with a block of $k$ don't cares*. Such repeats consist of two exact parts separated by a gap of fixed length $k$. Hence, our aim is to find all such repeats and their positions in the string.

⋆ Partially supported by CNRS, Wellcome Foundation, and Nato grants.

⋆⋆ Partially supported by a Marie Curie fellowship, Wellcome Foundation, Nato and Royal Society grants.

⋆⋆⋆ Supported by an EPSRC studentship

† Partially supported by French Programme BioInformatique Inter EPST, Wellcome Foundation, Royal Society and Nato grants.

A closely related problem was studied by Brodal *et al.* [2]. They developed algorithms for finding all "maximal pairs with bounded gap". This notion refers to a non extendable substring having two occurrences within a bounded distance of each other. A restricted version of the same problem was considered by Kolpakov and Kucherov [7]. They proposed an algorithm for a fixed gap. The problem of finding longest repeats with no don't cares is a mere application of suffix trees [5].

In our method we use two suffix trees intensively, one for the original string and the other for its reverse. The use of a generalized 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 [9] but there are some doubt that it will lead to a significant improvement on the asymptotic time complexity.

The paper is organized as follows: in Section 2, we state the preliminaries used throughout the paper. In Section 3, we define the longest repeat with $k$ don't cares and describe in general how to find them using two suffix trees. In Section 4, we detail our algorithm. Finally in Section 5, we analyze the running time of the algorithm.

## 2 Preliminaries

Throughout the paper $x$ denotes a *string* of length $n$ defined on a finite alphabet $\Sigma$. We use $x[i]$, for $i = 1, 2, \ldots, n$, to denote the $i$-th letter of $x$, and $x[i..j]$ as a notation for the *substring* $x[i]x[i+1]\cdots x[j]$ of $x$. The string $\overleftarrow{x}$ denotes the *reverse* of $x$, such that $\overleftarrow{x}[1] = x[n], \ldots, \overleftarrow{x}[n] = x[1]$.

The *length* of a string $w$ is denoted by $|w|$. If $w = uv$ then $w$ is said to be the *concatenation* of the two strings $u$ and $v$. The string $w^k$ is the $k$-th power of $w$.

A symbol '$\diamond$' $\notin \Sigma$ is called a "don't care"; any other symbol is called *solid*. A don't care *matches* any other symbol, that is, $\diamond = \sigma$ for each $\sigma \in \Sigma \cup \{\diamond\}$. A pattern $y$ over $\Sigma \cup \{\diamond\}$ is said to *occur* in $x$ at position $i$ if $y[j] = x[i+j-1]$, for $1 \leq j \leq |y|$. A *motif* $w$ denotes a pattern that occurs at least twice in $x$. We restrict the motifs to have a solid symbol at both ends, i.e., $w[1] \neq \diamond$ and $w[|w|] \neq \diamond$. The set $\mathcal{L}_w$ is the set of occurrence positions of a given motif $w$, where $\mathcal{L}_w = \{x[i..i+|w|-1], 1 \leq i \leq n-|w|+1\}$. Observe that $|\mathcal{L}_w| \geq 2$.

For a given string $x$ and an integer $k$, a motif $w$ of the form $L \diamond^k R$ is called *repeat with $k$ don't cares*. The substrings $L$ and $R$, respectively, are the left and right parts of $w$. The length of the longest such repeat in $x$ is denoted by $lr_k(x)$. Later on, we use the following notion: a motif $w$ is called *left maximal* (resp. *right maximal*) if $w$ can not be extended to the left (resp. right) without losing one of its occurrences.

Here we present a method for finding all longest repeats with $k$ contiguous don't cares and their positions. This 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 [5] or [4]. However, for the sake of completeness, we will briefly review the notion.

**Definition 1 (Suffix tree).** *The suffix tree $\mathcal{T}(x)$ of the string $x$ is the compacted trie of all suffixes of $x\$$, where $\$ \notin \Sigma$. Each leaf in $\mathcal{T}(x)$ represents a suffix $x[i..n]$ of $x$ and is labelled with the index $i$. We refer to the set of indices stored at the leaves of the subtree rooted at node $v$ as the leaf-list of $v$; it is denoted by $LL(v)$. Each edge in $\mathcal{T}(x)$ is labelled with a nonempty substring of $x$ such that the path from the root to the leaf labelled with index $i$ spells the suffix $x[i..n]$. We refer to the substring of $x$ spelled by the path from the root to a node $v$ as the label of $v$, and denote it by $\ell_v$. The length of such a substring is the depth of $v$ and we denote it by $d_v$.*

Several algorithms construct the suffix tree $\mathcal{T}(x)$ in $O(n)$ time, assuming an alphabet of fixed size (see for example [4] [5]). All the internal nodes in $\mathcal{T}(x)$ have an out-degree between 2 and $|\Sigma|$. Therefore, we can transform the suffix tree into a binary suffix tree $\mathcal{B}(x)$ 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 the binary suffix tree $\mathcal{B}(x)$ 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 the Schieber and Vishkin [8] *Lowest Common Ancestor* algorithm. For a given rooted tree $T$, the *lowest common ancestor* of two nodes $u$ and $v$, $lca(u, v)$, is the deepest node in $T$ that is ancestor of both $u$ and $v$. After a linear-time preprocessing of a rooted tree, the lowest common ancestor of any two nodes can be found in constant time.

## 3 Longest Repeats with $k$ Don't Cares

The *longest repeats with $k$ don't cares* problem requires finding all longest repeats of the form $L \diamond^k R$, that appear in a given string $x$. In the notation, $L$ and $R$ are both over $\Sigma$ and represent the left and the right parts, respectively, of the repeat. The parameter $k$ is a given positive integer smaller than $n$. For example, if

$$x = BBAZYABAAAXBBAXZABAZAHIABAA$$

then the only longest repeat with 2 don't cares is $w = BBA \diamond \diamond ABA$ and its occurrence list is $\mathcal{L}_w\{1, 12\}$. Thus, $lr_2(x) = 8$ . An obvious approach to solve this problem is as follows:

1. generate all possible repeated substrings in $x$;
2. for each pair of repeated substrings $u$ and $v$, check whether there exist at least two pairs of occurrence positions $i_1$ and $i_2$ of $u$ and $j_1$ and $j_2$ of $v$ such that $j_l = i_l + |u| + k$;
3. calculate the length of the repeat with $k$ don't cares $u \diamond^k v$;
4. report all longest ones.

This straightforward approach can be improved by dynamic programming yielding an $O(n^2)$ time algorithm.

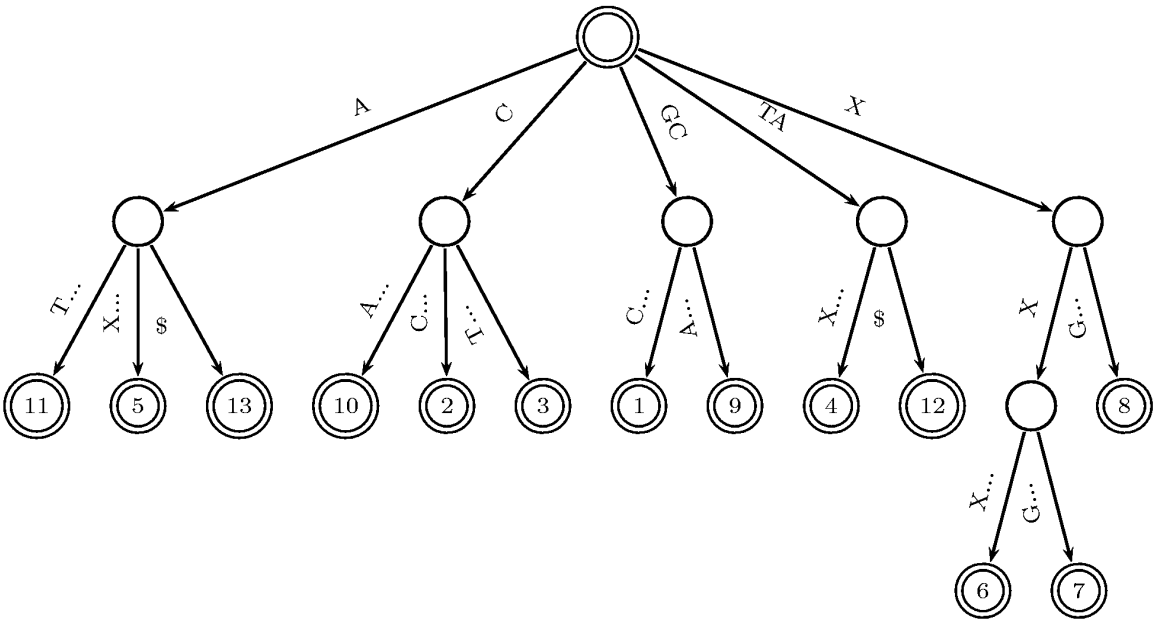


**Fig. 1.** The suffix tree of `GCCTAXXXGCATA`.

Our approach proceeds differently and results in an $O(n \log n)$ time algorithm. It starts by constructing the two suffix trees $\mathcal{T}(x)$ and $\mathcal{T}(\overleftarrow{x})$. The first suffix tree is used to generate the right part of the repeat, while the second suffix tree generates the left part. Observe that the label $\ell_u$ of each internal node $u \in \mathcal{T}(x)$ represents a right-maximal repeated substring of $x$ which occurs at $LL(u)$. Similarly, the label $\ell_v$ for each internal node $v \in \mathcal{T}(\overleftarrow{x})$ represents the reverse of a left-maximal repeated substring of $x$ ending at positions $\{j \mid j = n + 1 - i, i \in LL(v)\}$.

For simplicity, we replace each index $i$ in $\mathcal{T}(\overleftarrow{x})$ by $n+1-i+(k+1)$. Our goal now, is to traverse both trees efficiently to find all pairs of nodes $u$ and $v$ where $u \in \mathcal{T}(x)$, $v \in \mathcal{T}(\overleftarrow{x})$, $|LL(u) \cap LL(v)| \geq 2$, and $d_u + d_v$ is maximum. For each pair $u$ and $v$, the concatenation of the reverse of the label of $v$, $k$ don't cares, and the label of $u$ gives a longest repeat with $k$ don't cares, i.e., $w = \overleftarrow{\ell_v} \diamond^k \ell_u$. Observe that, $lr_k = d_v + k + d_u$.

For example, if $x =$ `GCCTAXXXGCATA` and $k = 1$, then Fig. 1 and Fig. 2 represent the suffix trees of, respectively, $x$ and $\overleftarrow{x}$. Note that, each index $i$ in $\mathcal{T}(\overleftarrow{x})$ has been replaced by $16 - i$. The node in $\mathcal{T}(x)$ labelled by $TA$ and node in $\mathcal{T}(\overleftarrow{x})$ labelled by $CG$ both have leaf-list $\{4, 12\}$. Thus, $GC \diamond TA$ is a repeat with 1 don't care. Since it is the longest such repeat, $lr_1(x)$ equals 5. The list of occurrence positions of the longest repeat with one don't care of $x$ is $\{1, 9\}$.

## 4 Algorithm

The initialization phase of the algorithm consists of two main steps. In the first step, the suffix tree of $x$ is constructed and then traversed in a preorder manner

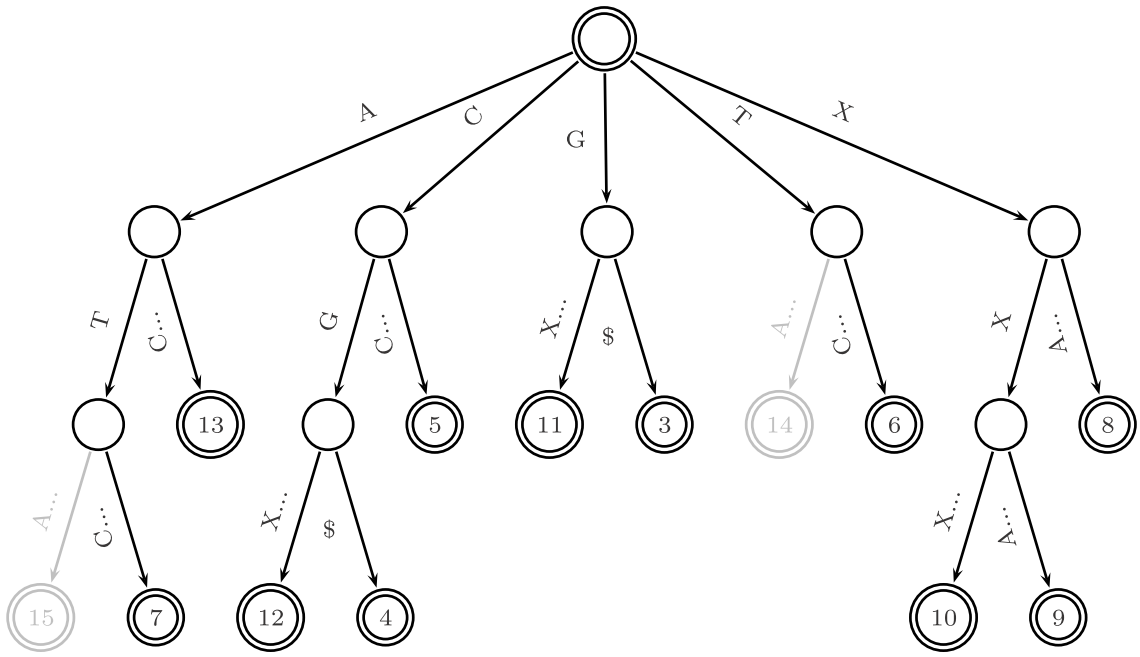


**Fig. 2.** The suffix tree of `ATACGXXXATCCG`. Each index $i$ is replaced by $16 - i$. The gray nodes may be omitted.

where a number is assigned to each node. For each index $i$, $no(i)$ is the preorder number assigned to the leaf node $v$ labelled with $i$ in $\mathcal{T}(x)$. This is done during the tree depth-first traversal. 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $no(i)$ | 11 | 8 | 9 | 14 | 4 | 18 | 19 | 20 | 12 | 7 | 3 | 15 | 5 |

In the second step, the suffix tree of $\overleftarrow{x}$ is built. In addition, a list is associated with each leaf node $v$. For each leaf node $v$ labelled with the $i$-th suffix of $\overleftarrow{x}$, this list is initialized with the element $no(n + 1 - i + (k - 1))$.

For each internal node $v$ , the list is the sorted union of the disjoint lists of the children of $v$. The computation of the lists for the internal nodes can be done during a depth-first traversal of the tree. However, in order to guarantee an efficient merge of the lists associated with the children of a node, $\mathcal{T}(\overleftarrow{x})$ is transformed into a binary suffix tree $\mathcal{B}(\overleftarrow{x})$. Furthermore, to maintain these lists efficiently, these lists were implemented using AVL-trees [1]. Although this implementation is similar to the one used in [2] and [6], any other type of balanced search trees may be used. Note that 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 [3] are summarized in the following lemmas.

**Lemma 1.** *Two AVL trees of size at most $n$ and $m$ can be merged in time $O(\log \binom{n+m}{n})$.*

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

*Proof.* The basic idea is to use the merge algorithm of Lemma 1 while keeping the positions where the insertions of the elements $e_i \in T$ take place. This change in the merge algorithm does not affect the time complexity and as a result we can find all $q_i$ in $O(\log \binom{n+m}{n})$ time.

Using the *smaller-half trick*, which states that *"the sum over all nodes $v$ of an arbitrary binary tree of terms that are $O(n_1)$, where $n_1$ and $n_2$ are the numbers of leaves in the subtrees rooted at the children of $v$ and $n_1 \leq n_2$, is $O(n \log n)$"*, the following lemma stated in [2] is easy to prove:

**Lemma 3.** *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 $O(n \log n)$.*

The algorithm for finding all longest repeats with $k$ don't cares is given in Fig. 3. Recall that at every node $v$ in $\mathcal{B}(\overleftarrow{x})$ we construct a sorted list, stored in an AVL tree $\mathcal{A}$, of all the preorder numbers associated with the elements in $LL(v)$. This list can be considered as a leaf-list sorted according to the preorder numbers associated to the indices in $\mathcal{T}(x)$. If $v$ is a leaf, then $\mathcal{A}$ is constructed directly (Line 5). If $v$ is an internal node, then $\mathcal{A}$ is constructed by merging $\mathcal{A}_1$ and $\mathcal{A}_2$ (Line 27), where $\mathcal{A}_1$ and $\mathcal{A}_2$ are the AVL trees associated with the two children of $v$ and $|\mathcal{A}_1| \leq |\mathcal{A}_2|$. Before constructing $\mathcal{A}$, we use $\mathcal{A}_1$ and $\mathcal{A}_2$ to check for an occurrence of longest repeat with $k$ don't cares. If a number $a$ in $\mathcal{A}_1$ is going to be inserted between $b$ and $c$ in $\mathcal{A}_2$, then $b$ and $c$ are efficiently obtained (Lemma 2). Let $max$ be the length of the current longest repeat with $k$ don't cares. And let $u$ and $v$ be the nodes representing this longest repeat, where $u \in \mathcal{T}(x)$ and $v \in \mathcal{B}(\overleftarrow{x})$. Since we are moving upward in $\mathcal{B}(\overleftarrow{x})$ minimizing the depth of $v$, the only way to find a longer repeat with $k$ don't cares is by replacing node $u$ in $\mathcal{T}(x)$ with a node that has greater depth. Clearly, this node should be a lowest common ancestor of a pair of nodes that has not been considered so far, *i.e.* a pair consisting of an element in $\mathcal{A}_1$ and an element in $\mathcal{A}_2$. It follows from Lemma 4, that we do not need to consider all the possible new pairs. In other words, only the pairs of the form (a,b) or (a,c) are the ones that need to be considered. For each pair of nodes considered by the algorithm, the algorithm checks whether the sum of the depth of both nodes is greater than or equal to $max$. If so, the algorithm uses list $M$ to store the pair. Note that the longest repeat with $k$ don't cares may not be unique. So, each pair $(x, y)$ in $M$ represents a longest repeat obtained by a concatenation of $\overleftarrow{\ell_y}$, $k$ don't cares, and $\ell_x$. Where $lr_k(x)$ equals $d_x + k + d_y$ for all pairs $(x, y) \in M$.

**Lemma 4.** *Let $i$, $j$ and $k$ be the preorder numbers given to three leaves $u$, $v$ and $w$ during a preorder traversal of a rooted tree $T$. If $i < j < k$, then the depth of $lca(u, v)$ cannot be less than the depth of $lca(u, w)$, where $lca$ is the lowest common ancestor of two nodes.*

**Algorithm** *Longest-Repeat-Don't-Cares*$(x, k)$
**Input:** A string $x$ of length $n$
**Output:** All longest repeats with $k$ contiguous don't cares

```
Build the suffix tree T(x) and traverse the tree in preorder manner numbering all
    the nodes.
for each leaf v ∈ T(x)
        if v is labelled with i
           then no(i) ← the preorder number of v
Build the binary suffix tree B(x̄) and create at each leaf an AVL tree of size one
    that stores no(n + 1 − i + (k + 1)), where i is the index associated with the leaf.
(max, u, v) ← (0, root(T(x)), root(B(x̄)))
M ←∅
for each node v ∈ B(x̄) in bottom-up (depth-first) manner
        A1, A2 ← the AVL trees of the two children of v where |A1| ≤ |A2|
        for a ∈ A1 in ascending order
                b ← max{x ∈ A2 | x ≤ a}
                ab ← lca(no^{-1}(a), no^{-1}(b)) in T(x)
                if d_ab + d_v = max
                   then (u, v) ←  (ab, v)
                        M ←M ∪ (ab, v)
                   else  if d_ab + d_v > max
                            then (max, u, v) ←  (d_ab + d_v, ab, v)
                                 M ←(ab, v)
                c ← next(T2, b)
                ac ← lca(no^{-1}(a), no^{-1}(c)) in T(x)
                if d_ac + d_v = max
                   then (u, v) ←  (ac, v)
                        M←M ∪ (ac, v)
                   else  if d_ac + d_v > max
                            then (max, u, v) ←  (d_ac + d_v, ac, v)
                                 M←(ac, v)
        A ←  merge(A1, A2)
lr_k ←max + k
return (lr_k, M)
```

**Fig. 3.** All longest repeats with $k$ don't cares algorithm.

*Proof.* The proof is by contradiction. Let $x$ and $y$ be $lca(u, v)$ and $lca(u, w)$, respectively. Assume that the depth of $x$ is less than the depth of $y$. Since $i$ is less than $j$, $k$ also must be less than $j$, which contradicts the condition that $i < j < k$.

The depth of a node in the Lemma 4 is the length of the path from the root to this node. It is quite easy to see that the Lemma can be extended to suffix trees where the depth of a node is the length of the substring spelled by the path from the root to this node.

## 5 Time Complexity

In this section, we analyze the running time of the algorithm. Recall that, for constant size alphabet, a suffix tree can be built in linear time. Thus, Creating $\mathcal{T}(x)$ and performing the preorder traversal at Line 1 requires $O(n)$ time. The loop on Lines 2-4 takes $O(n)$ time. Building $\mathcal{B}(\overleftarrow{x})$ also takes $O(n)$ time. Creating an AVL tree of size one can be done in constant time. Thus, doing so at each of the $n$ leaves of $\mathcal{B}(\overleftarrow{x})$ at Line 5 requires total of $O(n)$ time. Lines 6,7 take $O(1)$ time.

The algorithm then traverses $\mathcal{B}(\overleftarrow{x})$ in depth-first manner (Lines 8-27). At every internal node $v$, the algorithm runs a search loop on Lines 10-26 and then performs a merge at Line 27. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be the two AVL trees associated with the two children of $v$ where $|\mathcal{A}_1| \leq |\mathcal{A}_2|$. During the search loop (Lines 10-26), for each $a \in \mathcal{A}_1$, the algorithm searches $\mathcal{A}_2$ to find $b$ and $c$. According to Lemma 2, the time required to complete the search loop at each node is $O(\log \binom{|\mathcal{A}_1|+|\mathcal{A}_2|}{|\mathcal{A}_1|})$. Additionally, Lemma 1 states that the merge at Line 27 takes also $O(\log \binom{|\mathcal{A}_1|+|\mathcal{A}_2|}{|\mathcal{A}_1|})$ time. Summing these terms over all the internal nodes of $\mathcal{B}(\overleftarrow{x})$ gives the total running time of the tree traversal (Lines 8-27), that is $O(n \log n)$ (Lemma 3). Thus, the total running time of the algorithm is $O(n \log n)$ time. The following theorem states the result.

**Theorem 1.** *Algorithm Longest-Repeats-Don't-Cares extracts all longest repeats with $k$ don't cares from a given string in $O(n \log n)$ time.*

## References


1. Adel'son-Vel'skii, G.M., Landis, Y.M.: An Algorithm for the Organisation of Information. Doklady Akademii Nauk SSSR, Vol. 146 (1962) 263–266
2. Brodal, G.S., Lyngsø, R.B., Pedersen, C.N.S., Stoye, J.: Finding Maximal Pairs with Bounded Gaps. Journal of Discrete Algorithms,, Special Issue of Matching Patterns, Vol. 1 **1** (2000) 77–104
3. Brown, M.R., Tarjan, R.E.: A Fast Merging Algorithm. Journal of the ACM, Vol. 26 **2** (1979) 211–226
4. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific (2002)
5. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge Univesity Press (1997)
6. Iliopoulos, C.S., Markis, C., Sioutas, S., Tsakalidis, A., Tsichlas, K.: Identififying Ocuurences of Maximal Pairs in Multiple Strings. CPM (2002) 133–143. LNCS Vol. 2373 (2002) 133–143
7. Kolpakov, R., Kucherov, G.: Finding Repeats with Fixed Gap. SPIRE (2002) 162–168
8. Schieber, B., Vishkin, U.: On Finding Lowest Common Ancestors: Simplifications and Parallization. SIAM Journal of Computation, Vol. 17 (1988) 1253–1262
9. Stoye, J.: Affix Trees. Diploma Thesis, Universität Bielefeld, Forschungsbericht der Technischen Fakultät, Abteilung Informationstechnik, Report 2000–04 (2000) (ISSN 0946-7831)