Finding maximal pairs with bounded gap

Gerth Stølting Brodal*, Rune B. Lyngsø*,
Christian N. S. Pedersen*, and Jens Stoye**

Abstract. A pair in a string is the occurrence of the same substring twice. A pair is maximal if the two occurrences of the substring cannot be extended to the left and right without making them different. The gap of a pair is the number of characters between the two occurrences of the substring. In this paper we present methods for finding all maximal pairs under various constraints on the gap. In a string of length n we can find all maximal pairs with gap in an upper and lower bounded interval in time $O(n \log n + z)$ where z is the number of reported pairs. If the upper bound is removed the time reduces to $O(n+z)$. Since a tandem repeat is a pair where the gap is zero, our methods can be seen as a generalization of finding tandem repeats. The running time of our methods equals the running time of well known methods for finding tandem repeats.

1 Introduction

A pair in a string is the occurrence of the same substring twice. A pair is left-maximal (right-maximal) if the characters to the immediate left (right) of the two occurrences of the substring are different. A pair is maximal if it is both left- and right-maximal. The gap of a pair is the number of characters between the two occurrences of the substring. For example, the two occurrences of the substring *ma* in the string *maximal* form a maximal pair of *ma* with gap two.

Gusfield [9, Sect. 7.12.3] describes how to report all maximal pairs in a string using the suffix tree of the string in time $O(n + z)$ and space $O(n)$, where n is the length of the string and z is the number of reported pairs. Since there is no restriction on the gap of the maximal pairs reported by this algorithm, many of them probably describe occurrences of substrings that are overlapping or far apart in the string. In many applications in computational biology this is unfortunate, so several papers address the problem of finding occurrences of similar substrings not too far apart [13, 17, 23].

In this paper we will describe how to find all maximal pairs in a string with gap in an upper and lower bounded interval in time $O(n \log n+z)$ and space $O(n)$. The interval of allowed gaps can be chosen such that we report a maximal pair

* Basic Research in Computer Science (BRICS), Centre of the Danish National Research Foundation, Department of Computer Science, University of Aarhus, Ny Munkegade, 8000 Århus C, Denmark. E-mail: {gerth,rlyngsoe,cstorm}@brics.dk. Supported by the ESPRIT Long Term Research Programme of the EU under project number 20244 (ALCOM-IT).

** Deutsches Krebsforschungszentrum (DKFZ), Theoretische Bioinformatik, Im Neuenheimer Feld 280, 69120 Heidelberg, Germany. E-mail: j.stoye@dkfz-heidelberg.de

only if the gap is between constants c_1 and c_2, but more generally, it can be chosen such that we report a maximal pair of α only if the gap is between $g_1(|\alpha|)$ and $g_2(|\alpha|)$, where g_1 and g_2 are functions that can be computed in constant time. This, for example, makes it possible to find all maximal pairs with gap between zero and some fraction of the length of the repeated substring. If we remove the upper bound on allowed gaps, and only require the gap of a reported pair of α to be at least $g_1(|\alpha|)$, then the running time reduces to $O(n+z)$. The methods we present all use the suffix tree as the fundamental data structure combined with efficient methods for merging search trees and heap-ordered trees.

The problem of finding occurrences of repeated substrings in a string is well studied. Most of the work has been concerned with efficient methods for finding occurrences of contiguously repeated substrings. An occurrence of a substring of the form $\alpha\alpha$ is called an occurrence of a square or a tandem repeat. Most well-known methods for finding the occurrences of all tandem repeats in a string require time $O(n \log n + z)$, where n is the length of the string and z is the number of reported occurrences of tandem repeats [5, 2, 18, 15, 24]. Work has also been done on just detecting whether or not a string contains a tandem repeat [19, 6]. Recently, extending on the idea presented in [6], two methods have been presented that find a compact representation of all tandem repeats in a string in time $O(n)$ [14, 10]. Other papers consider the problem of finding occurrences of contiguous repeats of substrings that are within some Hamming- or edit-distance of each other [16].

In biological sequence analysis searching for tandem repeats is used to reveal structural and functional information [9, pp. 139–142], but searching for exact tandem repeats can be too restrictive because of sequencing and other experimental errors. By searching for maximal pairs with small gaps (maybe depending on the length of the substring) it could be possible to compensate for these errors. On the other hand, finding maximal pairs with a gap within an interval can be seen as a generalization of finding occurrences of tandem repeats. Stoye and Gusfield [24] say that an occurrence of the tandem repeat $\alpha\alpha$ is a branching occurrence of the tandem repeat $\alpha\alpha$ if and only if the characters to the immediate right of the two occurrences of α are different, and they explain how to deduce the occurrence of all tandem repeats in a string from the occurrences of branching tandem repeats in time proportional to the number of tandem repeats. Since a branching occurrence of a tandem repeat is just a right-maximal pair with gap zero, the methods presented in this paper can be used to find all tandem repeats in time $O(n \log n + z)$. This matches the time bounds of previous published methods for this problem [5, 2, 18, 15, 24].

The rest of this paper is organized as follows. In Sect. 2 we define pairs and suffix trees and describe how in general to find pairs using the suffix tree. In Sect. 3 we present facts about efficient merging of search trees, and use them to formulate methods for finding all maximal pairs in a string with gap in an upper and lower bounded interval. In Sect. 4 we briefly discuss how to find all maximal pairs in a string with gap in a lower bounded interval. Finally, in Sect. 5 we summarize our work and discuss open problems.

2 Preliminaries

Throughout this paper S will denote a string of length n over a finite alphabet Σ. We will use $S[i]$, for $i = 1, 2, \ldots, n$, to denote the ith character of S, and use $S[i..j]$ as notation for the substring $S[i]S[i+1]\cdots S[j]$ of S. To be able to refer to the characters to the left and right of every character in S without worrying about the first and last character, we define $S[0]$ and $S[n+1]$ to be two distinct characters not appearing anywhere else in S.

In order to formulate methods for finding repetitive structures in S, we need a proper definition of such structures. An obvious definition is to find all pairs of identical substrings in S. This, however, leads to a lot of redundant output, e.g. in the string that consists of n identical characters there are $\Theta(n^3)$ such pairs. To limit the redundancy without sacrificing any meaningful structures Gusfield [9] defines maximal pairs.

Definition 1 (Pair). We say that $(i, j, |\alpha|)$ is a pair of α in S formed by i and j if and only if $1 \leq i < j \leq n - |\alpha| + 1$ and $\alpha = S[i..i+|\alpha|-1] = S[j..j+|\alpha|-1]$. The pair is left-maximal (right-maximal) if the characters to the immediate left (right) of two occurrences of α are different, i.e. left-maximal if $S[i-1] \neq S[j-1]$ and right-maximal if $S[i+|\alpha|] \neq S[j+|\alpha|]$. The pair is maximal if it is right- and left-maximal. The gap of a pair $(i, j, |\alpha|)$ is the number of characters $j - i - |\alpha|$ between the two occurrences of α in S.

It follows from the definition that a string of length n in the worst case contains $\Theta(n^2)$ right-maximal pairs. The string a^n contains the worst case number of right-maximal pairs but only $\Theta(n)$ maximal pairs. The string $(aab)^{n/3}$ however contains $\Theta(n^2)$ maximal pairs. This shows that the worst case number of maximal pairs and right-maximal pairs in a string are asymptotically equal.

Figure 1 illustrates the occurrence of a pair. In some applications it might be interesting only to find pairs that obey certain restrictions on the gap, e.g. to filter out pairs of substrings that are overlapping or far apart and thus to reduce the number of pairs to report. Using the "smaller-half trick", see Sect. 3.1, and Lemma 3 it is easy to prove that a string of length n in the worst case contains $\Theta(n \log n)$ right-maximal pairs with gap in an interval of constant size.

In this paper we present methods for finding all right-maximal and maximal pairs $(i, j, |\alpha|)$ in S with gap in a bounded interval. These methods all use the suffix tree of S as the fundamental data structure. We briefly review the suffix tree and refer to [9] for a more comprehensive treatment.

Definition 2 (Suffix tree). The suffix tree $T(S)$ of the string S is the compressed trie of all suffixes of S. Each leaf in $T(S)$ represents a suffix $S[i..n]$ of S and is annotated with the index i. We refer to the set of indices stored at the leaves in the subtree rooted at node v as the leaf-list of v and denote it $LL(v)$. Each edge in $T(S)$ is labelled with a nonempty substring of S such that the path from the root to the leaf annotated with index i spells the suffix $S[i..n]$. We refer to the substring of S spelled by the path from the root to node v as the path-label of v and denote it $L(v)$.

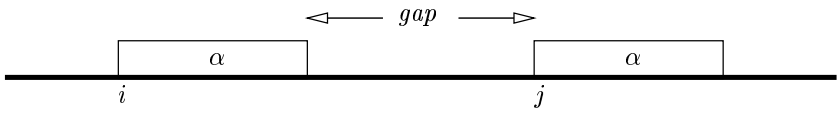

Fig. 1. An occurrence of a pair $(i, j, |\alpha|)$ with gap $j - i - |\alpha|$.

The suffix tree $T(S)$ can be constructed in time $O(n)$ [26, 20, 25, 7]. It follows from the definition that all internal nodes in $T(S)$ have out-degree between two and $|\Sigma|$. We can turn the suffix tree $T(S)$ into the binary suffix tree $T_B(S)$ by replacing every node v in $T(S)$ with out-degree $d > 2$ by a binary tree with $d-1$ internal nodes and $d-2$ internal edges in which the d leaves are the d children of node v. We label each new internal edge with the empty string such that the $d-1$ nodes replacing node v all have the same path-label as node v has in $T(S)$. Since $T(S)$ has n leaves, constructing the binary suffix tree $T_B(S)$ requires adding at most $n-2$ new nodes. Since each new node can be added in constant time, the binary suffix tree $T_B(S)$ can be constructed in time $O(n)$.

The binary suffix tree is an essential component of our methods. Definition 2 implies that there is a node v in $T(S)$ with path-label α if and only if α is the longest common prefix of $S[i\,..\,n]$ and $S[j\,..\,n]$ for some $1 \leq i < j \leq n$. In other words, there is a node v with path-label α if and only if $(i, j, |\alpha|)$ is a right-maximal pair in S. Since $S[i + |\alpha|] \neq S[j + |\alpha|]$ the indices i and j cannot be elements in the leaf-list of the same child of v. Using the binary suffix tree $T_B(S)$ we can thus formulate the following lemma.

Lemma 3. *There is a right-maximal pair $(i, j, |\alpha|)$ in S if and only if there is a node v in the binary suffix tree $T_B(S)$ with path-label α and distinct children w_1 and w_2 where $i \in LL(w_1)$ and $j \in LL(w_2)$.*

Lemma 3 gives an approach to find all right-maximal pairs in S; for every internal node v in the binary suffix tree $T_B(S)$ consider the leaf-lists at its two children w_1 and w_2, and for every element (i, j) in $LL(w_1) \times LL(w_2)$ report a right-maximal pair $(i, j, |\alpha|)$ if $i < j$ and $(j, i, |\alpha|)$ if $j < i$. To find all maximal pairs in S the problem remains to filter out all right-maximal pairs that are not left-maximal.

3 Pairs with upper and lower bounded gap

We want to find all maximal pairs $(i, j, |\alpha|)$ in S with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$, i.e. $g_1(|\alpha|) \leq j - i - |\alpha| \leq g_2(|\alpha|)$, where g_1 and g_2 are functions that can be computed in constant time. An obvious approach is to generate all maximal pairs in S and only report those with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$, but as shown above there might be asymptotically fewer maximal pairs in S with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ than maximal pairs in S in total. We

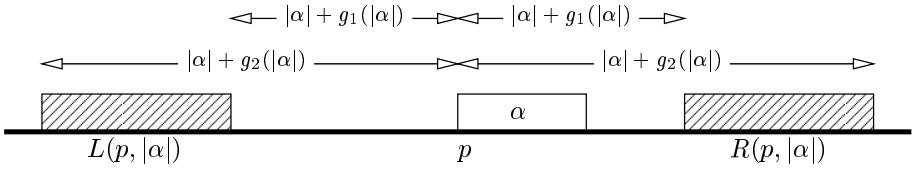

Fig. 2. If $(p, q, |\alpha|)$ (respectively $(q, p, |\alpha|)$) is a pair with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$, then one occurrence of α is at position p and the other occurrence is at a position q in the interval $R(p, |\alpha|)$ (respectively $L(p, |\alpha|)$) of positions.

therefore want to find all maximal pairs $(i, j, |\alpha|)$ in S with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ *without* generating and considering all maximal pairs in S. A step towards finding all maximal pairs with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ is to find all right-maximal pairs with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$.

Figure 2 shows that if one occurrence of α in a pair with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ is at position p, then the other occurrence of α must be at a position q in one of the two intervals $L(p, |\alpha|) = [\, p - |\alpha| - g_2(|\alpha|) \,..\, p - |\alpha| - g_1(|\alpha|) \,]$ or $R(p, |\alpha|) = [\, p + |\alpha| + g_1(|\alpha|) \,..\, p + |\alpha| + g_2(|\alpha|) \,]$. Together with Lemma 3 this gives an approach to find all right-maximal pairs in S with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$; from every internal node v in the binary suffix tree $T_B(S)$ with path-label α and children w_1 and w_2, we report for every p in $LL(w_1)$ the pairs $(p, q, |\alpha|)$ for all q in $LL(w_2) \cap R(p, |\alpha|)$ and the pairs $(q, p, |\alpha|)$ for all q in $LL(w_2) \cap L(p, |\alpha|)$.

To report right-maximal pairs efficiently using this procedure, we must be able to find for every p in $LL(w_1)$, without looking at all the elements in $LL(w_2)$, the proper elements q in $LL(w_2)$ to report it against. It turns out that search trees make this possible. In this paper we use AVL trees, but other types of search trees, e.g. (a, b)-trees [11] or red-black trees [8], can also be used as long as they obey Lemmas 4 and 5 stated below. Before we can formulate algorithms we review some useful facts about AVL trees.

3.1 Data Structures

An AVL tree T is a balanced search tree that stores an ordered set of elements. AVL trees were introduced in [1], but are explained in almost every textbook on data structures. We say that an element e is in T, or $e \in T$, if it is stored at a node in T. For short notation we use e to denote both the element and the node at which it is stored in T. We can keep links between the nodes in T in such a way that we in constant time from the node e can find the nodes $next(e)$ and $prev(e)$ storing the next and previous element in increasing order. We use $|T|$ to denote the size of T, i.e. the number of elements stored in T.

Efficient merging of two AVL trees is essential to our methods. Hwang and Lin [12] show how to merge two sorted lists using the optimal number of com-

parisons. Brown and Tarjan [4] show how to implement merging of two height-balanced search trees, e.g. AVL trees, in time proportional to the optimal number of comparisons. Their result is summarized in Lemma 4, which immediately implies Lemma 5.

Lemma 4. *Two AVL trees of size at most n and m can be merged in time $O(\log \binom{n+m}{n})$.*

Lemma 5. *Given a sorted list of elements $e_1, e_2, \ldots, e_n$ and an AVL tree T of size at most m, $m \geq n$, we can find $q_i = \min\{x \in T \mid x \geq e_i\}$ for all $i = 1, 2, \ldots, n$ in time $O(\log \binom{n+m}{n})$.*

Proof. Construct the AVL tree of the elements $e_1, e_2, \ldots, e_n$ in time $O(n)$. Merge this AVL tree with T according to Lemma 4, except that whenever the merge-algorithm would insert one of the elements $e_1, e_2, \ldots, e_n$ into T, we change the merge-algorithm to report the neighbor of the element in T instead. This modification does not increase the running time. □

The "smaller-half trick" is essential to several methods for finding tandem repeats [5, 2, 24]. It says that the sum over all nodes v in an arbitrary binary tree of size n of terms that are $O(n_1)$, where $n_1 \leq n_2$ are the numbers of leaves in the subtrees rooted at the two children of v, is $O(n \log n)$. Our methods rely on a stronger version of the "smaller-half trick" hinted at in [21, Ex. 35] and used in [22, Chap. 5, p. 84]; we summarize it in the following lemma.

Lemma 6. *Let T be an arbitrary binary tree with n leaves. The sum over all internal nodes v in T of terms that are $O(\log \binom{n_1+n_2}{n_1})$, where n_1 and n_2 are the numbers of leaves in the subtrees rooted at the two children of v, is $O(n \log n)$.*

Proof. As the terms are $O(\log \binom{n_1+n_2}{n_1})$ we can find constants, a and b, such that the terms are upper bounded by $a + b \log \binom{n_1+n_2}{n_1}$. We will by induction in the number of leaves of the binary tree prove that the sum is upper bounded by $(2n-1)a + b \log n!$. As $\log n! = O(n \log n)$ the lemma follows.

If T is a leaf then the upper bound holds vacuously. Now assume inductively that the upper bound holds for all trees with at most $n-1$ leaves. Let T be a tree with n leaves where the number of leaves in the subtrees rooted at the two children of the root are $n_1 < n$ and $n_2 < n$. According to the induction hypothesis the sum over all nodes in these two subtrees, i.e. the sum over all nodes of T except the root, is bounded by $(2n_1 - 1)a + b \log n_1! + (2n_2 - 1)a + b \log n_2!$ and thus the entire sum is bounded by

$$\begin{aligned}(2n_1 - 1)a + b \log n_1! + (2n_2 - 1)a + b \log n_2! + a + b \log \binom{n_1 + n_2}{n_1} \\ = (2(n_1 + n_2) - 1)a + b \log n_1! + b \log n_2! + \\ b \log(n_1 + n_2)! - b \log n_1! - b \log n_2! \\ = (2n - 1)a + b \log n!\end{aligned}$$

which proves the lemma. □

3.2 Algorithms

We first describe an algorithm that finds all right-maximal pairs in S with bounded gap using AVL trees to keep track of the elements in the leaf-lists during a traversal of the binary suffix tree $T_B(S)$. We then extend it to find all maximal pairs in S with bounded gap using an additional AVL tree to filter out efficiently all right-maximal pairs that are not left-maximal. Both algorithms run in time $O(n \log n + z)$ and space $O(n)$, where z is the number of reported pairs. In the following we assume, unless stated otherwise, that v is a node in the binary suffix tree $T_B(S)$ with path-label α and children w_1 and w_2 named such that $|LL(w_1)| \leq |LL(w_2)|$. We say that w_1 is the small child of v and that w_2 is the big child of v.

Right-maximal pairs with upper and lower bounded gap To find all right-maximal pairs in S with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ we consider every node v in the binary suffix tree $T_B(S)$ in a bottom-up fashion, e.g. during a depth-first traversal. At every node v we use AVL trees storing the leaf-lists $LL(w_1)$ and $LL(w_2)$ at its two children to report the proper right-maximal pairs of its path-label α. The details are given in Algorithm 1 and explained below.

At every node v in $T_B(S)$ we construct an AVL tree, the leaf-list tree T, that stores the elements in $LL(v)$. If v is a leaf then we construct T directly in Step 1. If v is an internal node then $LL(v)$ is the union of the disjoint leaf-lists $LL(w_1)$ and $LL(w_2)$ which by assumption are stored in the already constructed T_1 and T_2, so we construct T by merging T_1 and T_2, $|T_1| \leq |T_2|$, using Lemma 4. Before constructing T in Step 2c we use T_1 and T_2 to report right-maximal pairs from node v by reporting every p in $LL(w_1)$ against every q in $LL(w_2) \cap L(p, |\alpha|)$ and $LL(w_2) \cap R(p, |\alpha|)$. This is done in two steps. In Step 2a we find for every p in $LL(w_1)$ the minimum element $q_r(p)$ in $LL(w_2) \cap R(p, |\alpha|)$ and the minimum element $q_l(p)$ in $LL(w_2) \cap L(p, |\alpha|)$ by searching in T_2 using Lemma 5. In Step 2b we report pairs $(p, q, |\alpha|)$ and $(q, p, |\alpha|)$ for every p in $LL(w_1)$ and increasing q's in $LL(w_2)$ starting with $q_r(p)$ and $q_l(p)$ respectively, until the gap violates the upper or lower bound.

To argue that Algorithm 1 finds all right-maximal pairs with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ it is enough to argue that we for every p in $LL(w_1)$ report all right-maximal pairs $(p, q, |\alpha|)$ and $(q, p, |\alpha|)$ with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$. The rest follows because we at every node v in $T_B(S)$ consider every p in $LL(w_1)$. Consider the call $\mathsf{Report}(q_r(p), p + |\alpha| + g_2(|\alpha|))$ in Step 2b. From the implementation of Report follows that this call reports p against every q in $LL(w_2) \cap [q_r(p) \,..\, p + |\alpha| + g_2(|\alpha|)]$. By construction of $q_r(p)$ and definition of $R(p, |\alpha|)$ follows that $LL(w_2) \cap [q_r(p) \,..\, p + |\alpha| + g_2(|\alpha|)]$ is equal to $LL(w_2) \cap R(p, |\alpha|)$, so the call reports all pairs $(p, q, |\alpha|)$ with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$. Similarly we can argue that the call $\mathsf{Report}(q_l(p), p - |\alpha| - g_1(|\alpha|))$ reports all pairs $(q, p, |\alpha|)$ with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$.

Now consider the running time of Algorithm 1. Building the binary suffix tree $T_B(S)$ and creating an AVL tree of size one at each leaf in Step 1 takes time $O(n)$. At every internal node in $T_B(S)$ we do Step 2. Since $|T_1| \leq |T_2|$

Algorithm 1 Find all right-maximal pairs in string S with bounded gap.

1. *Initializing:* Build the binary suffix tree $T_B(S)$ and create at each leaf an AVL tree of size one that stores the index at the leaf.
2. *Reporting and merging:* When the AVL trees T_1 and T_2, $|T_1| \leq |T_2|$, at the two children w_1 and w_2 of node v with path-label α are available, we do the following:
   (a) Let $\{p_1, p_2, \ldots, p_s\}$ be the elements in T_1 in sorted order. For each element p in T_1 we find

$$q_r(p) = \min\{x \in T_2 \mid x \geq p + |\alpha| + g_1(|\alpha|)\}$$
$$q_l(p) = \min\{x \in T_2 \mid x \geq p - |\alpha| - g_2(|\alpha|)\}$$

   by searching in T_2 with the sorted lists $\{p_i + |\alpha| + g_1(|\alpha|) \mid i = 1, 2, \ldots, s\}$ and $\{p_i - |\alpha| - g_2(|\alpha|) \mid i = 1, 2, \ldots, s\}$ using Lemma 5.

   (b) For each element p in T_1 we do $\mathsf{Report}(q_r(p), p + |\alpha| + g_2(|\alpha|))$ and $\mathsf{Report}(q_l(p), p - |\alpha| - g_1(|\alpha|))$ where Report is the following procedure.

```
def Report(from, to) :
    q = from
    while q ≤ to :
        report pair (p, q, |α|) if p < q, and (q, p, |α|) otherwise
        q = next(q)
```

   (c) Build the leaf-list tree T at node v by merging T_1 and T_2 using Lemma 4.

searching in Step 2a and merging in Step 2c takes time $O(\log \binom{|T_1|+|T_2|}{|T_1|})$ by Lemmas 5 and 4 respectively. Reporting of pairs in Step 2b takes time proportional to $|T_1|$, because we consider every p in $LL(w_1)$, plus the number of reported pairs. Summing this over all nodes gives by Lemma 6 that the total running time is $O(n \log n + z)$, where z is the number of reported pairs. Since constructing and keeping $T_B(S)$ requires space $O(n)$, and since no element at any time is in more than one leaf-list tree, Algorithm 1 requires space $O(n)$.

Theorem 7. *Algorithm 1 finds all right-maximal pairs $(i, j, |\alpha|)$ in a string S with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ in space $O(n)$ and time $O(n \log n + z)$, where z is the number of reported pairs and n is the length of S.*

Maximal pairs with upper and lower bounded gap We now turn towards finding all maximal pairs in S with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$. Our approach to find all maximal pairs in S with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ is to extend Algorithm 1 to filter out all right-maximal pairs that are not left-maximal. A simple solution is to extend the procedure Report to check if $S[p-1] \neq S[q-1]$ before reporting the pair $(p, q, |\alpha|)$ or $(q, p, |\alpha|)$ in Step 2b. This solution takes time proportional to the number of inspected right-maximal pairs, and not time proportional to the number of reported maximal pairs. Even though the maximum number of right-maximal pairs and maximal pairs in strings of a given

length are asymptotically equal, many strings contain significantly fewer maximal pairs than right-maximal pairs. We therefore want to filter out all right-maximal pairs that are not left-maximal *without* inspecting all right-maximal pairs. In the remainder of this section we describe one way to do this.

Consider the reporting step in Algorithm 1 and assume that we are about to report from a node v with children w_1 and w_2. The leaf-list trees T_1 and T_2, $|T_1| \leq |T_2|$, are available and they make it possible to access the elements in $LL(w_1) = \{p_1, p_2, \ldots, p_s\}$ and $LL(w_2) = \{q_1, q_2, \ldots, q_t\}$ in sorted order. We divide the sorted leaf-list $LL(w_2)$ into blocks of contiguous elements such that the elements q_{i-1} and q_i are in the same block if and only if $S[q_{i-1}-1] = S[q_i-1]$. We say that we divide the sorted leaf-list into blocks of elements with equal left-characters. To filter out all right-maximal pairs that are not left-maximal we must avoid to report p in $LL(w_1)$ against any element q in $LL(w_2)$ in a block of elements with left-character $S[p-1]$. This gives the overall idea of the extended algorithm; we extend the reporting step in Algorithm 1 such that whenever we are about to report p in $LL(w_1)$ against q in $LL(w_2)$ where $S[p-1] = S[q-1]$ we skip all elements in the current block containing q and continue reporting p against the first element q' in the following block, which by the definition of blocks satisfies that $S[p-1] \neq S[q'-1]$.

To implement this extended reporting step efficiently we must be able to skip all elements in a block without inspecting each of them. We achieve this by constructing an additional AVL tree, the block-start tree, that keeps track of the blocks in the leaf-list. At each node v during the traversal of $T_B(S)$ we thus construct two AVL trees; the leaf-list tree T that stores the elements in $LL(v)$, and the block-start tree B that keeps track of the blocks in the sorted leaf-list by storing all the elements in $LL(v)$ that start a block. We keep links from the block-start tree to the leaf-list tree such that we in constant time can go from an element in the block-start tree to the corresponding element in the leaf-list tree. Figure 3 illustrates the leaf-list tree, the block-start tree and the links between them. Before we present the extended algorithm and explain how to use the block-start tree to efficiently skip all elements in a block, we first describe how to construct the leaf-list tree T and block-start tree B at node v from the leaf-list trees, T_1 and T_2, and block-start trees, B_1 and B_2, at its two children w_1 and w_2.

Since $LL(v)$ is the union of the disjoint leaf-lists $LL(w_1)$ and $LL(w_2)$ stored in T_1 and T_2 respectively, we can construct the leaf-list tree T by merging T_1 and T_2 using Lemma 4. It is more involved to construct the block-start tree B. The reason is that an element p_i that starts a block in $LL(w_1)$ or an element q_j that starts a block in $LL(w_2)$ does not necessarily start a block in $LL(v)$ and vice versa, so we cannot construct B by merging B_1 and B_2. Let $\{e_1, e_2, \ldots, e_{s+t}\}$ be the elements in $LL(v)$ in sorted order. By definition the block-start tree B contains all elements e_k in $LL(v)$ where $S[e_{k-1}-1] \neq S[e_k-1]$. We construct B by modifying B_2. We choose to modify B_2, and not B_1, because $|LL(w_1)| \leq |LL(w_2)|$, which by the "smaller-half trick" allows us to consider all elements in $LL(w_1)$ without spending too much time in total. To modify B_2 to become B we must identify all the elements that are in B but not in B_2 and vice versa.

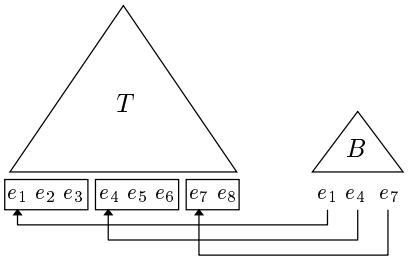

Fig. 3. The data structure constructed at each node v in $T_B(S)$. The leaf-list tree T stores all elements in $LL(v)$. The block-start tree B stores all elements in $LL(v)$ that start a block in the sorted leaf-list. We keep links from the elements in the block-start tree to the corresponding elements in the leaf-list tree.

Lemma 8. *If e_k is in B but not in B_2 then $e_k \in LL(w_1)$ or $e_{k-1} \in LL(w_1)$.*

Proof. Assume that e_k is in B and that e_k and e_{k-1} both are in $LL(w_2)$. In $LL(w_2)$ the elements e_k and e_{k-1} are neighboring elements q_j and q_{j-1}. Since e_k starts a block in $LL(v)$ then $S[q_j - 1] = S[e_k - 1] \neq S[e_{k-1} - 1] = S[q_{j-1} - 1]$. This shows that $q_j = e_k$ is in B_2 and the lemma follows. □

Let *NEW* be the set of elements e_k in B where e_k or e_{k-1} are in $LL(w_1)$. It follows from Lemma 8 that this set contains at least all elements in B that are not in B_2. It is easy to see that we can construct *NEW* in sorted order while merging T_1 and T_2; whenever an element e_k from T_1, i.e. $LL(w_1)$, is placed in T, i.e. $LL(v)$, we include it, and/or the next element e_{k+1} placed in T, in *NEW* if they start a block in $LL(v)$.

If we insert the elements in *NEW* we are halfway done modifying B_2 to become B. We still need to identify and remove the elements that should be removed from B_2, that is, the elements that are in B_2 but not in B.

Lemma 9. *An element q_j in B_2 is not in B if and only if the largest element e_k in NEW smaller than q_j in B_2 has the same left-character as q_j.*

Proof. If q_j is in B_2 but does not start a block in $LL(v)$, then it must be in a block started by some element e_k with the same left-character as q_j. This block cannot contain q_{j-1} because q_j being in B_2 implies that $S[q_j - 1] \neq S[q_{j-1} - 1]$. We thus have the ordering $q_{j-1} < e_k < q_j$. This implies that e_k is the largest element in *NEW* smaller than q_j. If e_k is the largest element in *NEW* smaller than q_j, then no block starts in $LL(v)$ between e_k and q_j, i.e. all elements e in $LL(v)$ where $e_k < e < q_j$ satisfy that $S[e-1] = S[e_k-1]$, so if $S[e_k-1] = S[q_j-1]$ then q_j does not start a block in $LL(v)$. □

By searching in B_2 with the sorted list *NEW* using Lemma 5 it is straightforward to find all pairs of elements (e_k, q_j), where e_k is the largest element in

NEW smaller than q_j in B_2. If the left-characters of e_k and q_j in such a pair are equal, i.e. $S[e_k - 1] = S[q_j - 1]$, then by Lemma 9 the element q_j is not in B and must therefore be removed from B_2. It follows from the proof of Lemma 9 that if this is the case then $q_{j-1} < e_k < q_j$, so we can, without destroying the order among the nodes in B_2, remove q_j from B_2 and insert e_k instead, simply by replacing the element q_j with the element e_k at the node storing q_j in B_2.

We can now summarize the three steps it takes to modify B_2 to become B. In Step 1 we construct the sorted set *NEW* that contains all elements in B that are not in B_2. This is done while merging T_1 and T_2 using Lemma 4. In Step 2 we remove the elements from B_2 that are not in B. The elements in B_2 being removed and the elements from *NEW* replacing them are identified using Lemmas 5 and 9. In Step 3 we merge the remaining elements in *NEW* into the modified B_2 using Lemma 4. Adding links from the new elements in B to the corresponding elements in T can be done while replacing and merging in Steps 2 and 3. Since $|NEW| \leq 2\,|T_1|$ and $|B_2| \leq |T_2|$, the time it takes to construct B is dominated by the the time it takes merge a sorted list of size $2\,|T_1|$ into an AVL tree of size $|T_2|$. By Lemma 4 this is within a constant factor of the time it takes to merge T_1 and T_2, so the time is takes to construct B is dominated by the time it takes to construct the leaf-list tree T.

Now that we know how to construct the leaf-list tree T and block-start tree B at node v from the leaf-list trees, T_1 and T_2, and block-start trees, B_1 and B_2, at its two children w_1 and w_2, we can proceed with the implementation of the extended reporting step. The details are shown in Algorithm 2. This algorithm is similar to Algorithm 1 except that we at every node v in $T_B(S)$ construct two AVL trees; the leaf-list tree T that stores the elements in $LL(v)$, and the block-start tree B that keeps track of the blocks in $LL(v)$ by storing the subset of elements that start a block. If v is a leaf, we construct T and B directly. If v is an internal node, we construct T by merging the leaf-list trees T_1 and T_2 at its two children w_1 and w_2, and we construct B by modifying the block-start tree B_2 as explained above.

Before constructing T and B we report all maximal pairs from node v with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ by reporting every p in $LL(w_1)$ against every q in $LL(w_2) \cap L(p, |\alpha|)$ and $LL(w_2) \cap R(p, |\alpha|)$ where $S[p-1] \neq S[q-1]$. This is done in two steps. In Step 2a we find for every p in $LL(w_1)$ the minimum elements $q_l(p)$ and $q_r(p)$, as well as the minimum elements $b_l(p)$ and $b_r(p)$ that start a block, in $LL(w_2) \cap L(p, |\alpha|)$ and $LL(w_2) \cap R(p, |\alpha|)$ respectively. This is done by searching in T_2 and B_2 using Lemma 5. In Step 2b we report pairs $(p, q, |\alpha|)$ and $(q, p, |\alpha|)$ for every p in $LL(w_1)$ and increasing q's in $LL(w_2)$ starting with $q_r(p)$ and $q_l(p)$ respectively, until the gap violates the upper or lower bound. Whenever we are about to report p against q where $S[p-1] = S[q-1]$, we instead use the block-start tree B_2 to skip all elements in the block containing q and continue with reporting p against the first element in the following block.

To argue that Algorithm 2 finds all maximal pairs with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ it is enough to argue that we for every p in $LL(w_1)$ report all maximal pairs $(p, q, |\alpha|)$ and $(q, p, |\alpha|)$ with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$. The rest

Algorithm 2 Find all maximal pairs in string S with bounded gap.

1. *Initializing:* Build the binary suffix tree $T_B(S)$ and create at each leaf two AVL trees of size one, the leaf-list and the block-start tree, both storing the index at the leaf.
2. *Reporting and merging:* When the leaf-list trees T_1 and T_2, $|T_1| \leq |T_2|$, and the block-start trees B_1 and B_2 at the two children w_1 and w_2 of node v with path-label α are available, we do the following:
   (a) Let $\{p_1, p_2, \ldots, p_s\}$ be the elements in T_1 in sorted order. For each element p in T_1 we find

$$\begin{aligned} q_r(p) &= \min\{x \in T_2 \mid x \geq p + |\alpha| + g_1(|\alpha|)\} \\ q_l(p) &= \min\{x \in T_2 \mid x \geq p - |\alpha| - g_2(|\alpha|)\} \\ b_r(p) &= \min\{x \in B_2 \mid x \geq p + |\alpha| + g_1(|\alpha|)\} \\ b_l(p) &= \min\{x \in B_2 \mid x \geq p - |\alpha| - g_2(|\alpha|)\} \end{aligned}$$

   by searching in T_2 and B_2 with the sorted lists $\{p_i + |\alpha| + g_1(|\alpha|) \mid i = 1, 2, \ldots, s\}$ and $\{p_i - |\alpha| - g_2(|\alpha|) \mid i = 1, 2, \ldots, s\}$ using Lemma 5.
   (b) For each element p in T_1 we do $\mathsf{ReportMax}(q_r(p), b_r(p), p + |\alpha| + g_2(|\alpha|))$ and $\mathsf{ReportMax}(q_l(p), b_l(p), p - |\alpha| - g_1(|\alpha|))$ where ReportMax is the following procedure.

```
def ReportMax(from_T, from_B, to):
    q = from_T
    b = from_B
    while q ≤ to:
        if S[q − 1] ≠ S[p − 1]:
            report pair (p, q, |α|) if p < q, and (q, p, |α|) otherwise
            q = next(q)
        else:
            while b ≤ q:
                b = next(b)
            q = b
```

   (c) Build the leaf-list tree T at node v by merging T_1 and T_2 using Lemma 4. Build the block-start tree B at node v by modifying B_2 as described in the text.

follows because we at every node in $T_B(S)$ consider every p in $LL(w_1)$. Consider the call $\mathsf{ReportMax}(q_r(p), b_r(p), p + |\alpha| + g_2(|\alpha|))$ in Step 2b. From the implementation of ReportMax follows that unless we skip elements by increasing b then we consider every q in $LL(w_2) \cap R(p, |\alpha|)$. The test $S[q-1] \neq S[p-1]$ before reporting a pair ensures that we only report maximal pairs and whenever $S[q-1] = S[p-1]$ we increase b until $b = \min\{x \in B_2 \mid x > q\}$. This is, by construction of B_2 and $b_r(p)$, the element that starts the block following the block containing q, so all elements q', $q < q' < b$, we skip by setting q to b satisfy that $S[p-1] = S[q-1] = S[q'-1]$. We thus conclude that

$\mathsf{ReportMax}(q_r(p), b_r(p), p + |\alpha| + g_2(|\alpha|))$ reports p against exactly those q in $LL(w_2) \cap R(p, |\alpha|)$ where $S[p-1] \neq S[q-1]$, i.e. it reports all maximal pairs $(p, q, |\alpha|)$ at node v with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$. Similarly, the call $\mathsf{ReportMax}(q_l(p), b_l(p), p - |\alpha| - g_1(|\alpha|))$ reports all maximal pairs $(q, p, |\alpha|)$ with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$.

Now consider the running time of Algorithm 2. We first argue that the call $\mathsf{ReportMax}(q_r(p), b_r(p), p + |\alpha| + g_2(|\alpha|))$ takes constant time plus time proportional to the number of reported pairs $(p, q, |\alpha|)$. To do this all we have to show is that the time used to skip blocks, i.e. the number of times we increase b, is proportional to the number of reported pairs. By construction $b_r(p) \geq q_r(p)$, so the number of times we increase b is bounded by the number of blocks in $LL(w_2) \cap R(p, |\alpha|)$. Since neighboring blocks contain elements with different left-characters, we report p against an element from at least every second block in $LL(w_2) \cap R(p, |\alpha|)$. The number of times we increase b is thus proportional to the number of reported pairs. The call $\mathsf{ReportMax}(q_l(p), b_l(p), p - |\alpha| - g_1(|\alpha|))$ also takes constant time plus time proportional to the number of reported pairs $(q, p, |\alpha|)$. We thus have that Step 2b takes time proportional to $|T_1|$ plus the number of reported pairs. Everything else we do at node v, i.e. searching in T_2 and B_2 and constructing the leaf-list tree T and block-start tree B, takes time $O(\log \binom{|T_1|+|T_2|}{|T_1|})$. Summing this over all nodes gives by Lemma 6 that the total running time of the algorithm is $O(n \log n + z)$ where z is the number of reported pairs. Since constructing and keeping $T_B(S)$ requires space $O(n)$, and since no element at any time is in more than one leaf-list tree, and maybe one block-start tree, Algorithm 2 requires space $O(n)$.

Theorem 10. *Algorithm 2 finds all maximal pairs $(i, j, |\alpha|)$ in a string S with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ in space $O(n)$ and time $O(n \log n + z)$, where z is the number of reported pairs and n is the length of S.*

We observe that Algorithm 2 never uses the block-start tree B_1 at the small child w_1. This observation can be used to ensure that only one block-start tree exists during the execution of the algorithm. If we implement the traversal of $T_B(S)$ as a depth-first traversal in which we at each node v first recursively traverse the subtree rooted at the small child w_1, then we do not need to store the block-start tree returned by this recursive traversal while recursively traversing the subtree rooted at the big child w_2. This implies that only one block-start tree exists at all times during the recursive traversal of $T_B(S)$. The drawback is that we at each node v need to know in advance which child is the small child, but this knowledge can be obtained in linear time by annotating each node with the size of the subtree it roots.

4 Pairs with lower bounded gap

If we relax the constraint on the gap and only want to find all maximal pairs in S with gap at least $g(|\alpha|)$, where g is a function that can be computed in constant time, then a straightforward solution is to use Algorithm 2 with

$g_1(|\alpha|) = g(|\alpha|)$ and $g_2(|\alpha|) = n$. This obviously finds all maximal pairs with gap at least $g_1(|\alpha|) = g(|\alpha|)$ in time $O(n \log n + z)$. However, the missing upper bound on the gap, i.e. the trivial upper bound $g_2(|\alpha|) = n$, makes it possible to reduce the running time to $O(n + z)$ since reporting from each node during the traversal of the binary suffix tree is simplified.

The reporting of pairs from node v with children w_1 and w_2 is simplified, because the lack of an upper bound on the gap implies that we do not have to search $LL(w_2)$ for the first element to report against the current element in $LL(w_1)$. Instead we can start by reporting the current element in $LL(w_1)$ against the biggest (and smallest) element in $LL(w_2)$ and then continue reporting it against decreasing (and increasing) elements from $LL(w_2)$ until the gap becomes smaller than $g(|\alpha|)$. Unfortunately this simplification alone does not reduce the asymptotic running time because inspecting every element in $LL(w_1)$ and keeping track of the leaf-lists in AVL trees alone requires time $\Theta(n \log n)$. To reduce the running time we must thus avoid to inspect every element in $LL(w_1)$ and find another way to store the leaf-lists.

We achieve this by using a data structure based on heap-ordered trees to store the leaf-lists during the traversal of the binary suffix tree. The key feature of the data structure is that it allows us to merge two trees in amortized constant time. The details of the data structure and the methods using it to find pairs with gap at least $g(|\alpha|)$ is given in [3, Sect. 4]. Here we just summarize the result.

Theorem 11. *All maximal pairs $(i, j, |\alpha|)$ in a string S with gap at least $g(|\alpha|)$ can be found in space $O(n)$ and time $O(n+z)$, where z is the number of reported pairs and n is the length of S.*

5 Conclusion

We have presented efficient and flexible methods to find all maximal pairs $(i, j, |\alpha|)$ in a string under various constraints on the gap $j - i - |\alpha|$. If the gap is required to be between $g_1(|\alpha|)$ and $g_2(|\alpha|)$, the running time is $O(n \log n + z)$ where n is the length of the string and z is the number of reported pairs. If the gap is only required to be at least $g_1(|\alpha|)$, the running time reduces to $O(n+z)$. In both cases we use space $O(n)$.

In some cases it might be interesting only to find maximal pairs $(i, j, |\alpha|)$ fulfilling additional requirements on $|\alpha|$, e.g. to filter out pairs of short substrings. This is straightforward to do using our methods by only reporting from the nodes in the binary suffix tree whose path-label α fulfills the requirements on $|\alpha|$. In other cases it might be of interest just to find the vocabulary of substrings that occur in maximal pairs. This is also straightforward to do using our methods by just reporting the path-label α of a node if we can report one or more maximal pairs from the node.

Instead of just looking for maximal pairs, it could be interesting to look for an array of occurrences of the same substring in which the gap between consecutive occurrences is bounded by some constants. This problem requires a

suitable definition of a maximal array. One definition and approach is presented in [23]. Another definition inspired by the definition of a maximal pair could be to require that every pair of occurrences in the array is a maximal pair. This definition seems very restrictive. A more relaxed definition could be to only require that we cannot extend all the occurrences in the array to the left or to the right without destroying at least one pair of occurrences in the array.

Acknowledgments This work was initiated while Christian N. S. Pedersen and Jens Stoye were visiting Dan Gusfield at UC Davis. We would like to thank Dan Gusfield, as well as Rob Irwing, for listening to some preliminary results.

References

1. G. M. Adel'son-Vel'skii and Y. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962. English translation in *Soviet Math. Dokl.*, 3:1259–1262.
2. A. Apostolico and F. P. Preparata. Optimal off-line detection of repetitions in a string. *Theoretical Computer Science*, 22:297–315, 1983.
3. G. S. Brodal, R. B. Lyngsø, C. N. S. Pedersen, and J. Stoye. Finding maximal pairs with bounded gap. Technical Report RS-99-12, BRICS, April 1999.
4. M. R. Brown and R. E. Tarjan. A fast merging algorithm. *Journal of the ACM*, 26(2):211–226, 1979.
5. M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981.
6. M. Crochemore. Tranducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.
7. M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 137–143, 1997.
8. L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 8–21, 1978.
9. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
10. D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. Technical Report CSE-98-4, Department of Computer Science, UC Davis, 1998.
11. S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.
12. F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal on Computing*, 1(1):31–39, 1972.
13. S. Karlin, M. Morris, G. Ghandour, and M.-Y. Leung. Efficient algorithms for molecular sequence analysis. *Proceedings of the National Academy of Science, USA*, 85:841–845, 1988.
14. R. Kolpakov and G. Kucherov. Maximal repetitions in words or how to find all squares in linear time. Technical Report 98-R-227, LORIA, 1998.
15. S. R. Kosaraju. Computation of squares in a string. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 807 of *Lecture Notes in Computer Science*, pages 146–150, 1994.

16. G. M. Landau and J. P. Schmidt. An algorithm for approximate tandem repeats. In *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 684 of *Lecture Notes in Computer Science*, pages 120–133, 1993.
17. M.-Y. Leung, B. E. Blaisdell, C. Burge, and S. Karlin. An efficient algorithm for identifying matches with errors in multiple long molecular sequences. *Journal of Molecular Biology*, 221:1367–1378, 1991.
18. M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5:422–432, 1984.
19. M. G. Main and R. J. Lorentz. Linear time recognition of squarefree strings. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pages 271–278. Springer, Berlin, 1985.
20. E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
21. K. Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1994.
22. K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999. To appear. See http://www.mpi-sb.mpg.de/~mehlhorn/LEDAbook.html.
23. M.-F. Sagot and E. W. Myers. Identifying satellites in nucleic acid sequences. In *Proceedings of the 2nd Annual International Conference on Computational Molecular Biology (RECOMB)*, pages 234–242, 1998.
24. J. Stoye and D. Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. In *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1448 of *Lecture Notes in Computer Science*, pages 140–152, 1998.
25. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.
26. P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Symposium on Switching and Automata Theory*, pages 1–11, 1973.