Finding Maximal Pairs with Bounded Gap

GERTH STØLTING BRODAL[*], *BRICS*[†], *Department of Computer Science, University of Aarhus, Ny Munkegade, 8000 Århus C, Denmark. E-mail: gerth@brics.dk*

RUNE BANG LYNGSØ, *Baskin Center for Computer Science and Engineering, University of California, Santa Cruz, CA 95064, USA. E-mail: rlyngsoe@cse.ucsc.edu*

CHRISTIAN NØRGAARD STORM PEDERSEN[*], *BRICS*[†], *Department of Computer Science, University of Aarhus, Ny Munkegade, 8000 Århus C, Denmark. E-mail: cstorm@brics.dk*

JENS STOYE, *DKFZ*[‡], *Theoretische Bioinformatik (H0300), Im Neuenheimer Feld 280, 69120 Heidelberg, Germany. E-mail: j.stoye@dkfz.de*

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, and 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 with gap zero, our methods is a generalization of finding tandem repeats. The running time of our methods also equals the running time of well known methods for finding tandem repeats.

Keywords: Strings, Maximal Pairs, Tandem Repeats, Suffix Trees, Efficient Merging, Search Trees

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

A preliminary version of this paper appeared in the Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching [3].

[*] Partially supported by the ESPRIT Long Term Research Programme of the EU under project number 20244 (ALCOM-IT).

[†] Basic Research in Computer Science, Centre of the Danish National Research Foundation.

[‡] Deutsches Krebsforschungszentrum (German Cancer Research Center)

gap of a pair is the number of characters between the two occurrences of the substring, e.g. the two occurrences of the substring *ma* in the string *maximal* form a maximal pair of *ma* with gap two. Gusfield in [11, Section 7.12.3] describes how to use a suffix tree to report all maximal pairs in a string of length n in time $O(n+z)$ and space $O(n)$, where z is the number of reported pairs. The algorithm presented by Gusfield allows no restrictions on the gaps of the reported maximal pairs, so many of the reported pairs probably describe occurrences of substrings that are either overlapping or far apart in the string. In many applications this is unfortunate because it leads to a lot of redundant output. The problem of finding occurrences of similar substrings not too far apart has been studied in several papers, e.g. [15, 19, 25].

In the first part of this paper we 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 only if the gap is between two constants c_1 and c_2; but more generally, the interval can be chosen such that we report a maximal pair 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 and $|\alpha|$ is the length of the repeated substring. 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. In the second part of this paper we describe how removing the upper bound $g_2(|\alpha|)$ on the allowed gaps makes it possible to reduce the running time to $O(n+z)$. The methods we present all use the suffix tree as the fundamental data structure combined with efficient merging of search trees and heap-ordered trees.

Finding occurrences of repeated substrings in a string is a widely studied problem. Much work has focused on constructing 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. Several methods have been presented that in time $O(n \log n + z)$ find all z occurrences of tandem repeats in a string of length n, e.g. [2, 5, 17, 20, 26]. Methods that in time $O(n)$ decide if a string of length n contains an occurrence of a tandem repeat have also been presented, e.g. [6, 21]. Extending on the ideas presented in [6], two methods [12, 16] have been presented that find a compact representation of all tandem repeats in a string of length n in time $O(n)$. The problem of finding occurrences of contiguous repeats of substrings that are within some Hamming- or edit-distance of each other is considered in e.g. [18].

In biological sequence analysis searching for tandem repeats is used to reveal structural and functional information [11, pp. 139–142]. However, 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. Finding maximal pairs with gap in a bounded interval is also a generalization of finding occurrences of tandem repeats. Stoye and Gusfield in [26] 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, e.g. [2, 5, 17, 20, 26].

The rest of this paper is organized in two parts which can be read independently. In Section 2 we present the preliminaries necessary to read either of the two parts; we define pairs and suffix trees and describe how in general to find pairs using the suffix tree. In the first part, Section 3, we present the methods to find all maximal pairs in a string with gap in an upper and lower bounded interval. This part also presents facts about efficient merging of search trees which are essential to the formulation of the methods. In the second part, Section 4, we present the methods to find all maximal pairs in a string with gap in a lower bounded interval. This part also includes the presentation of two novel data structures, the heap-tree and the colored heap-tree, which are essential to the formulation of the methods. Finally, in Section 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 meaningful structures Gusfield in [11] proposes maximal pairs.

DEFINITION 2.1 (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.

The indices i and j in a right-maximal pair $(i, j, |\alpha|)$ uniquely determine $|\alpha|$. Hence, a string of length n contains in the worst case $O(n^2)$ right-maximal pairs. The string a^n contains the worst case number of right-maximal pairs but only $O(n)$ maximal pairs. However, the string $(aab)^{n/3}$ 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

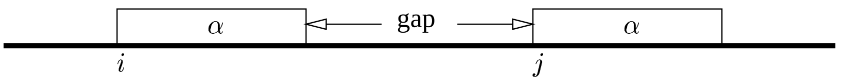

FIG. 1. An occurrence of a pair $(i, j, |\alpha|)$ with gap $j - i - |\alpha|$.

of substrings that are either overlapping or far apart and thus reduce the number of pairs to report. Using the "smaller-half trick" (see Section 3.1) and Lemma 2.3 it can be shown 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 [11] for a more comprehensive treatment.

DEFINITION 2.2 (Suffix tree)
The *suffix tree* $T(S)$ of the string S is the compressed trie of all suffixes of $S\$$, where $\$ \notin \Sigma$. Each leaf in $T(S)$ represents a suffix $S[i \mathinner{..} n]$ 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 $S[i \mathinner{..} n]$. 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 $L(v)$.

Several algorithms construct the suffix tree $T(S)$ in time $O(n)$, e.g. [7, 22, 28, 30]. It follows from the definition of a suffix tree 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 2.2 implies that there is an internal node v in $T(S)$ with path-label α if and only if α is the longest common prefix of $S[i \mathinner{..} n]$ and $S[j \mathinner{..} n]$ 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 2.3
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 $j \in LL(w_2)$.

The lemma implies 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 to solve this problem is to generate all maximal pairs in S but only report those with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$. However, as explained in the previous section 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 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:

$$L(p, |\alpha|) = [\, p - |\alpha| - g_2(|\alpha|) \,..\, p - |\alpha| - g_1(|\alpha|) \,] \quad (3.1)$$

$$R(p, |\alpha|) = [\, p + |\alpha| + g_1(|\alpha|) \,..\, p + |\alpha| + g_2(|\alpha|) \,] \quad (3.2)$$

Combined with Lemma 2.3 this gives an approach to find all right-maximal pairs in S with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$: for 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 the right-maximal pairs efficiently we must be able to find for every p in $LL(w_1)$ the proper elements q in $LL(w_2)$ to report it against, without looking at all the elements in $LL(w_2)$. 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 [13] or red-black trees [10], can also be used as long as they obey Lemmas 3.1 and 3.2 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

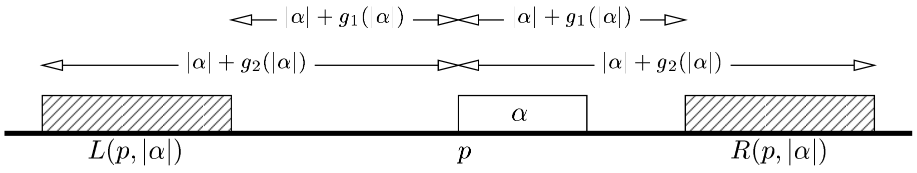
$|\alpha| + g_1(|\alpha|)$
$|\alpha| + g_1(|\alpha|)$
$|\alpha| + g_2(|\alpha|)$
$|\alpha| + g_2(|\alpha|)$
α
$L(p, |\alpha|)$
p
$R(p, |\alpha|)$

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.

element. 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 [14] show how to merge two sorted lists using the optimal number of comparisons. 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 3.1, which immediately implies Lemma 3.2.

LEMMA 3.1
Two AVL trees of size at most n and m can be merged in time $O(\log \binom{n+m}{n})$.

LEMMA 3.2
Given a sorted list of elements $e_1, e_2, \ldots, e_n$ and an AVL tree T of size at most m, where $m \geq n$, 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 3.1, 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 used in several methods for finding tandem repeats, e.g. [2, 5, 26]. 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 for finding maximal pairs rely on a stronger version of the "smaller-half trick" hinted at in [23, Exercise 35] and used in [24, Chapter 5, page 84]; we summarize it in the following lemma.

LEMMA 3.3
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 $O(n \log n)$.

PROOF. We will by induction in the number of leaves of the binary tree prove that the sum is upper bounded by $\log n!$. 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 $\log n_1! + \log n_2!$ and thus the entire sum is bounded by

$$\begin{aligned} &\log n_1! + \log n_2! + \log \binom{n_1+n_2}{n_1} \\ &= \log n_1! + \log n_2! + \log(n_1+n_2)! - \log n_1! - \log n_2! \\ &= \log n! \, . \end{aligned}$$

As $\log n! = O(n \log n)$ the lemma follows. ∎

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.

3.2.1 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 the two children of v to report the proper right-maximal pairs of the path-label α of v. The details are given in Algorithm 1 and explained next.

At every node v in $T_B(S)$ we construct an AVL tree, a *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 $LL(w_2)$. By assumption $LL(w_1)$ and $LL(w_2)$ are stored in the already constructed T_1 and T_2. We construct T by merging T_1 and T_2 using Lemma 3.1, where $|T_1| \leq |T_2|$. 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 $LL(w_2) \cap R(p, |\alpha|)$, where $L(p, |\alpha|)$ and $R(p, |\alpha|)$ are the intervals defined by (3.1) and (3.2). 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 $q_\ell(p)$ in $LL(w_2) \cap L(p, |\alpha|)$ by searching in T_2 using Lemma 3.2. 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 $q_\ell(p)$ 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 $g_2(|\alpha|)$ by Lemma 2.3 it is sufficient to show 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. The implementation of Report implies that p is reported against every q in $LL(w_2) \cap [q_r(p) \,..\, p + |\alpha| + g_2(|\alpha|)]$. The construction of $q_r(p)$ and the definition of $R(p, |\alpha|)$ implies that the set $LL(w_2) \cap [q_r(p) \,..\, p + |\alpha| + g_2(|\alpha|)]$ is equal to $LL(w_2) \cap R(p, |\alpha|)$. Hence, the call to Report reports all right-maximal pairs $(p, q, |\alpha|)$ with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$. Similarly the call $\mathsf{Report}(q_\ell(p), p - |\alpha| - g_1(|\alpha|))$ reports all right-maximal 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 $T_B(S)$ takes time $O(n)$ [7, 22, 28, 30], and creating an AVL tree of size one at each leaf in Step 1 also takes time $O(n)$. At every internal node in $T_B(S)$ we perform Step 2. Since $|T_1| \leq |T_2|$, the searching in Step 2a and the merging in Step 2c take time $O(\log \binom{|T_1|+|T_2|}{|T_1|})$ by Lemmas 3.2 and 3.1 respectively. The 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 3.3 that the total running time is $O(n \log n + z)$, where z is the number of reported pairs. Constructing and keeping $T_B(S)$ requires space $O(n)$. Since no element at any time is stored in more than one leaf-list tree, Algorithm 1 requires space $O(n)$ in total.

THEOREM 3.4
Algorithm 1 finds all right-maximal pairs $(i, j, |\alpha|)$ in a string S of length n 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.

3.2.2 Maximal pairs with upper and lower bounded gap

We now turn our attention towards finding all maximal pairs in S with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$. Our approach 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 approach to achieve this.

Consider the reporting step in Algorithm 1. Assume that we are about to report from a node v with children w_1 and w_2. At this point the leaf-list trees T_1 and T_2, where $|T_1| \leq |T_2|$, 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.

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, where $|T_1| \leq |T_2|$, at the two children w_1 and w_2 of a 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

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

   by searching in T_2 with the two 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 3.2.

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

```
Report($from$, $to$)
    $q = from$
    while $q \leq to$ do
        report pair $(p, q, |\alpha|)$ if $p < q$, and $(q, p, |\alpha|)$ otherwise
        $q = next(q)$
```

   (c) Build the leaf-list tree T at node v by merging T_1 and T_2 applying Lemma 3.1.

Our approach is to 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 $LL(v)$

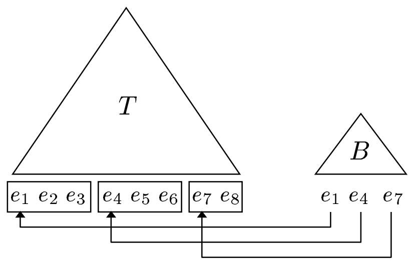

FIG. 3: The data structure constructed at each node v in $T_B(S)$. The leaf-list tree T stores all elements in $LL(v)$. The block-start tree B stores all elements in $LL(v)$ that start a block in the sorted leaf-list. We keep links from the elements in the block-start tree to the corresponding elements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

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 the block-start tree B at node v from the leaf-list trees, T_1 and T_2, and the block-start trees, B_1 and B_2, at its two children w_1 and w_2.

Since the leaf-list $LL(v)$ 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 3.1. 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.

LEMMA 3.5
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 both e_k and e_{k-1} are in $LL(w_2)$. In $LL(w_2)$ the elements e_k and e_{k-1} are neighboring elements. Let these neighboring elements in $LL(w_2)$ be denoted q_j and q_{j-1}. Since e_k is in B and therefore 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. ∎

In the following NEW denotes the set of elements e_k in B where either e_k or e_{k-1} is in $LL(w_1)$. It follows from Lemma 3.5 that NEW contains at least all elements in B that are not in B_2. We can construct NEW in sorted order while merging T_1 and T_2 as follows. When an element e_k from T_1, i.e. from $LL(w_1)$, is placed in T, i.e. in $LL(v)$, we include it in the set NEW if it starts a block in $LL(v)$. Similarly the next element e_{k+1} in $LL(v)$ is included in NEW if it starts a block in $LL(v)$.

Constructing the set NEW is the first step in modifying B_2 to become B. The next step is to identify the elements that should be removed from B_2, that is, to identify the elements that are in B_2 but not in B.

LEMMA 3.6
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 $LL(v)$. ■

To identify the elements that should be removed from B_2, we search B_2 with the sorted list NEW using Lemma 3.2 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 3.6 the element q_j is not in B and must therefore be removed from B_2. It follows from the proof of Lemma 3.6 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 3.1. 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 3.2 and 3.6. In Step 3 we merge the remaining elements in NEW into the modified B_2 using Lemma 3.1. 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 to merge a sorted list of size $2\,|T_1|$ into an AVL tree of size $|T_2|$. By Lemma 3.1 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

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, storing the index at the leaf.
2. *Reporting and merging:* When the leaf-list trees T_1 and T_2, where $|T_1| \leq |T_2|$, 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

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

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 3.2.

(b) For each element p in T_1 we call $\mathsf{ReportMax}(q_r(p), b_r(p), p + |\alpha| + g_2(|\alpha|))$ and $\mathsf{ReportMax}(q_\ell(p), b_\ell(p), p - |\alpha| - g_1(|\alpha|))$, where ReportMax is the following procedure.

```
ReportMax($from\_in\_T$, $from\_in\_B$, $to$)
    $q = from\_in\_T$
    $b = from\_in\_B$
    while $q \leq to$ do
        if $S[q-1] \neq S[p-1]$ then
            report pair $(p, q, |\alpha|)$ if $p < q$, and $(q, p, |\alpha|)$ otherwise
            $q = next(q)$
        else
            while $b \leq q$ do $b = next(b)$
            $q = b$
```

(c) Build the leaf-list tree T at node v by merging T_1 and T_2 using Lemma 3.1. Build the block-start tree B at node v by modifying B_2 as described in the text.

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 $q_\ell(p)$ and $q_r(p)$, as well as the minimum elements $b_\ell(p)$ 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 3.2. 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 $q_\ell(p)$ 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 the maximal pairs with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$ it is sufficient to show 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 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. The implementation of $\mathsf{ReportMax}$ implies that unless we skip elements by increasing b, we consider every q in $LL(w_2) \cap R(p, |\alpha|)$ exactly as in Algorithm 1. The test $S[q-1] \neq S[p-1]$ ensures that we only report maximal pairs. Whenever $S[q-1] = S[p-1]$ we increase b until $b = \min\{x \in B_2 \mid x > q\}$, which by construction of B_2 and $b_r(p)$ is the element that starts the block following the block containing q. Hence, all the elements q', where $q < q' < b$, we skip by setting q to b thus satisfy that $S[p-1] = S[q-1] = S[q'-1]$. We 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 $\mathsf{ReportMax}(q_\ell(p), b_\ell(p), p - |\alpha| - g_1(|\alpha|))$ reports all maximal pairs $(q, p, |\alpha|)$ with gap between $g_1(|\alpha|)$ and $g_2(|\alpha|)$.

We 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. Similarly the call $\mathsf{ReportMax}(q_\ell(p), b_\ell(p), p - |\alpha| - g_1(|\alpha|))$ 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 3.3 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 3.7
Algorithm 2 finds all maximal pairs $(i, j, |\alpha|)$ in a string S of length n 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.

As a closing remark we can 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 of $T_B(S)$ 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 $g(|\alpha|)$ in time $O(n \log n + z)$. However, the missing upper bound on the gap 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 the priority-queue like data structures presented in the next section to store the leaf-lists during the traversal of the binary suffix tree.

4.1 Data structures

A heap-ordered tree is a tree in which each node stores an element and has a key. Every node other than the root satisfies that its key is greater than or equal to the key at its parent. Heap-ordered trees have been widely studied and are the basic structure of many priority queues [8, 9, 29, 31]. In this section we utilize heap-ordered trees to construct two data structures, *the heap-tree* and *the colored heap-tree*, that are useful in our application of finding pairs with lower bounded gap but might also have applications elsewhere.

A heap-tree stores a collection of elements with comparable keys and supports the following operations.

$\mathsf{Init}(e,k)$:	Return a heap-tree of size one that stores element e with key k.
$\mathsf{Find}(H,x)$:	Return all elements e stored in the heap-tree H with key $k \leq x$.
$\mathsf{Min}(H)$:	Return the element e stored in H with minimum key.
$\mathsf{Meld}(H,H')$:	Return a heap-tree that stores all elements in H and H' with unchanged keys and colors.

A colored heap-tree stores a collection of colored elements with comparable keys. We use $color(e)$ to denote the color of element e. A colored heap-tree supports the same operations as a heap-tree except that it allows us to find all elements not having a particular color. The operations are as follows.

$\mathsf{ColorInit}(e,k)$:	Return a colored heap-tree of size one that stores element e with key k.
$\mathsf{ColorFind}(H,x,c)$:	Return all elements e stored in the colored heap-tree H with key $k \leq x$ and $color(e) \neq c$.
$\mathsf{ColorMin}(H)$:	Return the element e stored in H with minimum key.
$\mathsf{ColorSec}(H)$:	Return the element e stored in H with minimum key such that $color(e) \neq color(\mathsf{ColorMin}(H))$.
$\mathsf{ColorMeld}(H,H')$:	Return a colored heap-tree that stores all elements in H and H' with unchanged keys.

In the following we will describe how to implement heap-trees and colored heap-trees using heap-ordered trees such that Init, Min, ColorInit, ColorMin and ColorSec take constant time, Find and ColorFind take time proportional to the number of returned elements, and Meld and ColorMeld take amortized constant time. This means that we can meld n (colored) heap-trees of size one into a single (colored) heap-tree of size n by an arbitrary sequence of $n-1$ meld operations in time $O(n)$ in the worst case.

4.1.1 Heap-trees

We implement heap-trees as binary heap-ordered trees as illustrated in Figure 4. At every node in the heap-ordered tree we store an element from the collection of elements we want to store. The key of a node is the key of the element it stores. We use $v.elm$ to refer to the element stored at node v, $v.key$ to refer to the key of node v, and $v.right$ and $v.left$ to refer to the two children of node v. Besides the heap-order we maintain the invariant that the root of the heap-ordered tree has no left-child.

We define the *backbone* of a heap-tree as the path in the heap-ordered tree that starts at the root and continues via nodes reachable from the root via a sequence of right-children. We define the length of the backbone as the number of edges on the path it describes. Consider the heap-trees H and H' in Figure 4; the backbone of H is the path $r, v_1, \ldots, v_s$ of length s and the backbone of H' is the path $r', v'_1, \ldots, v'_t$ of length t. We say that the node on the backbone farthest from the root is at the bottom of the backbone. We keep track of the nodes on the backbone of a heap-tree using a stack, *the backbone-stack*, in which the root is at the bottom and the node farthest

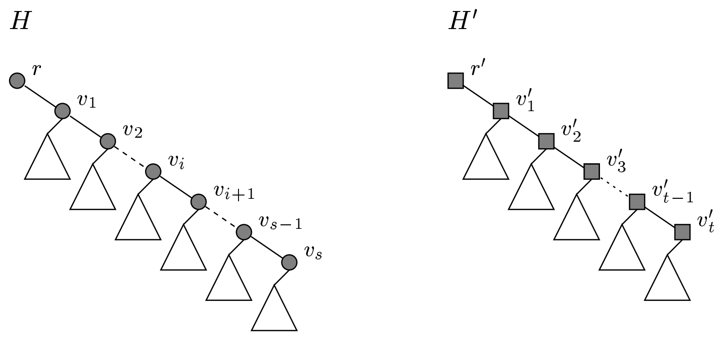

FIG. 4: Heap-trees are binary heap-ordered trees. The figure shows two heap-trees H and H'. The nodes on the backbone of the two heap-trees are shaded.

from the root is at the top. The backbone-stack makes it easy to access the nodes on the backbone from the bottom and up towards the root.

We now turn to the implementation of Init, Min, Find and Meld. The implementation of $\mathsf{Init}(e, k)$ is straightforward. We construct a single node v where $v.elm = e$, $v.key = k$ and $v.right = v.left = null$ and a backbone-stack of size one that contains node v. The implementation of $\mathsf{Min}(H)$ is also straightforward. The heap-order implies that root r of H stores the element with minimum key, i.e. $\mathsf{Min}(H) = r.elm$.

The implementation of $\mathsf{Find}(H, x)$ is based on a recursive traversal of H starting at the root. At each node v we compare $v.key$ to x. If $v.key \leq x$, we report $v.elm$ and continue recursively with the two children of v. If $v.key > x$, then by the heap-order all keys at nodes in the subtree rooted at v are greater than x, so we return from v without reporting. Clearly this traversal reports all elements stored at nodes v with $v.key \leq x$, i.e. all elements stored with key $k \leq x$. Since each node has at most two children, we make, for each reported element, at most two additional comparisons against x corresponding to the at most two recursive calls from which we return without reporting. The running time of the traversal is thus proportional to the number of reported elements.

The implementation of $\mathsf{Meld}(H, H')$ is done in two steps. Figure 5 illustrates the melding of the heap-trees H and H' from Figure 4. We assume that $r.key \leq r'.key$. In Step 1 we merge the backbones of H and H' together such that the heap-order is satisfied in the resulting tree. The merged backbone is constructed from the bottom and up towards the root by popping nodes from the backbone-stacks of H and H'. Step 1 results in a heap-tree with a backbone of length $s+t+1$. Since $r.key \leq r'.key$, a prefix of the merged backbone consists of nodes $r, v_1, v_2, \ldots, v_i$ solely from the backbone of H. In Step 2 we shorten the merged backbone. Since the root r' of H' has no left-child, the node r' on the merged backbone has no left-child either, so by moving the right-child of r' to this empty spot, making it the left-child of r', we shorten the length of the merged backbone to $i + 1$.

The two steps of $\mathsf{Meld}(H, H')$ clearly construct a heap-ordered tree that stores all

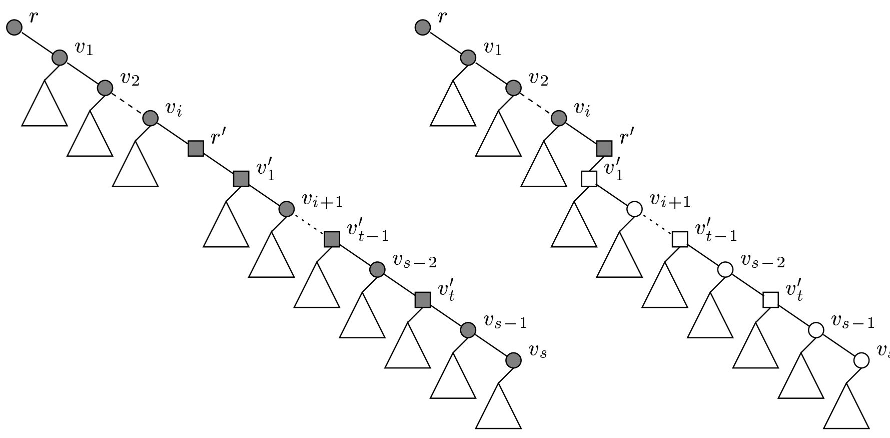

FIG. 5: The two steps of melding the heap-trees H and H' shown in Figure 4. The heap-tree to the left is the result of merging the backbones. The heap-tree to the right is the result of shortening the backbone by moving the right-child of r' in the merged backbone to the left-child. The nodes on the backbones are marked.

elements in H and H' with unchanged keys. Since $r.key \leq r'.key$, the root of the constructed heap-ordered tree is the root of H and therefore has no left-child. The constructed heap-ordered tree is thus a heap-tree as wanted. The backbone of the new heap-tree is the path $r, v_1, \ldots, v_i, r'$. We observe that the backbone-stack of H after Step 1 contains exactly the nodes $r, v_1, \ldots v_i$. We can thus construct the backbone-stack of the new heap-tree by pushing r' onto what remains of the backbone-stack of H after Step 1.

Now consider the running time of $\mathsf{Meld}(H, H')$. Step 1 takes time proportional to the total number of nodes popped from the two backbone-stacks. Since $i + 1$ nodes remains on the backbone-stack of H, Step 1 takes time $(s + 1) + (t + 1) - (i + 1) = s + t - i + 1$. Step 2 and construction of the new backbone-stack takes constant time, so, except for a constant factor, melding two heap-trees with backbones of length s and t takes time $T(s, t) = s + t - i + 1$. In our application of finding pairs we are more interested in bounding the total time required to do a sequence of melds rather than bounding the time of each individual meld. We therefore turn to amortized analysis [27].

On a forest F of heap-trees we define the potential function $\Phi(F)$ to be the sum of the lengths of the backbones of the heap-trees in the forest. Melding two heap-trees with backbones of length s and t, as illustrated in Figure 5, changes the potential of the forest with $\Delta\Phi = i + 1 - (s + t)$. The amortized running time of melding the two heap-trees is thus $T(s, t) + \Delta\Phi = (s + t - i + 1) + (i - s - t + 1) = 2$, so starting with n heap-trees of size one, i.e. a forest F_0 with potential $\Phi(F_0) = 0$, and doing a sequence of $n - 1$ meld operations until the forest F_{n-1} consists of a single heap-tree, takes time $O(n)$ in the worst case.

4.1.2 Colored heap-trees

We implement colored heap-trees as colored heap-ordered trees in much tqhe same way as we implemented heap-trees as uncolored heap-ordered trees. The implementation only differs in two ways. First, a node in the colored heap-ordered tree stores a set of elements instead of just a single element. Secondly, a node, including the root, can have several left-children. The elements stored at a node, and the references to the left-children of a node, are kept in uncolored heap-trees. More precisely, a node v in the colored heap-ordered tree has the following attributes.

$v.elms$: A heap-tree that stores the elements at node v. $\mathsf{Find}(v.elms, x)$ returns all elements stored at node v with key less than or equal to x. All elements stored at node v have identical colors. We say that this color is the color of node v and denote it by $color(v)$.

$v.key$: The key of node v. We set the key of a node to be the minimum key of an element stored at the node, i.e. the key of node v is the key of the element stored at the root of the heap-tree $v.elms$.

$v.right$: A reference to the right-child of node v.

$v.lefts$: A heap-tree that stores the references to the left-children of node v. A reference is stored with a key equal to the key of the referenced left-child, so $\mathsf{Find}(v.lefts, x)$ returns the references to all left-children of node v with key less than or equal to x.

As for the heap-tree we define the backbone of a colored heap-tree as the path that starts at the root and continues via nodes reachable from the root via a sequence of right-children. We use a stack, the backbone-stack, to keep track of the nodes on the backbone. In addition to the heap-order, saying that the key of every node other than the root is greater than or equal to the key of its parent, we maintain the following three invariants about the color of the nodes and the relation between the elements stored at a node and its left-children.

I_1: Every node v other than the root r has a color different from its parent.

I_2: Every node v satisfies that $|\mathsf{Find}(v.elms, x)| \geq |\mathsf{Find}(v.lefts, x)|$ for any x.

I_3: The root r satisfies that $|\mathsf{Find}(r.elms, x)| \geq |\mathsf{Find}(r.lefts, x)| + 1$ for any $x \geq \mathsf{Min}(r.elms)$.

We now turn our attention towards the implementation of the operations on colored heap-trees. $\mathsf{ColorInit}(e, k)$ is straightforward. We simply construct a single node v where $v.key = k$, $v.elms = \mathsf{Init}(e, k)$ and $v.right = v.lefts = null$ and a backbone-stack that contains node v. $\mathsf{ColorMin}(H)$ is also straightforward. The heap-order implies that the element with minimum key is stored in the heap-tree $r.elms$ at the root r of H, so $\mathsf{ColorMin}(H) = \mathsf{Min}(r.elms)$. The heap-order and I_1 imply that $\mathsf{ColorSec}(H)$ is the element stored with minimum key at a child of r. The element stored with minimum key at the right-child is $\mathsf{Min}(r.right)$ and the element stored with minimum key at a left-child must by the heap-order of $r.lefts$ be the element stored with minimum key at the left-child referenced by the root of $r.lefts$, i.e. $\mathsf{Min}(\mathsf{Root}(r.\mathit{lefts}).\mathit{elm})$. Both $\mathsf{ColorMin}(H)$ and $\mathsf{ColorSec}(H)$ can thus be found in constant time.

We implement $\mathsf{ColorFind}(H, x, c)$ as a recursive traversal of H starting at the root. More precisely, we implement $\mathsf{ColorFind}(H, x, c)$ as $\mathsf{ReportFrom}(r)$ where r is the root of H and ReportFrom is the following recursive procedure.

```
ReportFrom($v$)
    if $key(v) \leq x$ then
        if $color(v) \neq c$ then
            $E = \mathsf{Find}(v.elms, x)$
            for $e$ in $E$ do
                report $e$
        ReportFrom($v.right$)
        $W = \mathsf{Find}(v.\mathit{lefts}, x)$
        for $w$ in $W$ do
            ReportFrom($w$)
```

The correctness of this implementation is established as follows. The heap-order ensures that all nodes v with $v.key \leq x$ are visited during the traversal. The definition of $v.key$ implies that any element e with key $k \leq x$ is stored at a node v with $v.key \leq x$, i.e. among the elements returned by $\mathsf{Find}(v.elms, x)$ for some v visited during the traversal. Together with the test $color(v) \neq c$ this implies that all elements e with key $k \leq x$ and color different from c are reported by $\mathsf{ColorFind}(H, x, c)$.

Now consider the running time of $\mathsf{ColorFind}(H, x, c)$. Since $\mathsf{Find}(v.elms, x)$ and $\mathsf{Find}(v.\mathit{lefts}, x)$ both take time proportional to the number of returned elements, it follows that the running time is dominated by the number of recursive calls plus the number of reported elements. To argue that the running time of $\mathsf{ColorFind}(H, x, c)$ is proportional to the number of reported elements we therefore argue that the number of reported elements dominates the number of recursive calls. We only make recursive calls from a node v if $v.key \leq x$. Let v be such a node and consider two cases.

If $color(v) \neq c$ then we report at least one element, namely the element with key $v.key$, and by the invariants I_2 and I_3 we report at least as many elements as the number of left-children we call when reporting from v. Hence, except for a constant term that we can charge for visiting node v, the number of reported elements at v accounts for the call to v and all the recursive calls from v.

If $color(v) = c$ then we do not report any elements at v, but the invariant I_1 ensures that we have reported elements at its parent (unless v is the root) and that we will be reporting elements at all left-children we call from v. The call to v is thus already accounted for by the elements reported at its parent, and except for a constant term that we can charge for visiting node v, all calls from v will be accounted for by elements reported at the children of v. We conclude that the number of reported elements dominates the number of recursive calls, so $\mathsf{ColorFind}(H, x, c)$ takes time proportional to the number of reported elements.

We implement $\mathsf{ColorMeld}(H, H')$ similar to $\mathsf{Meld}(H, H')$ except that we must ensure that the constructed colored heap-tree obeys the three invariants. Let H and H' be colored heap-trees with roots r and r' named such that $r.key \leq r'.key$. We implement $\mathsf{ColorMeld}(H, H')$ as the following three steps.

1. *Merge*. We merge the backbones of H and H' together such that the resulting heap-ordered tree stores all elements in H and H' with unchanged keys. The merging is done by popping nodes from the backbone-stacks of H and H' until the backbone-stack of H' is empty
2. *Solve conflicts*. A node w on the merged backbone with the same color as its parent v is a violation of invariant I_1. We solve conflicts between neighboring nodes v and w of equal color by melding the elements and left-children of the two nodes and removing node w. We say that parent v swallows the child w.

   $v.elms = \mathsf{Meld}(v.elms, w.elms)$
   $v.lefts = \mathsf{Meld}(v.lefts, w.lefts)$
   $v.right = w.right$
3. *Shorten backbone*. Let v be the node on the merged backbone corresponding to r' or the node that swallowed r' in Step 2. We shorten the backbone by moving the right-child of v to the set of left-children of v.

   $v.lefts = \mathsf{Meld}(v.lefts, \mathsf{Init}(v.right, v.right.key))$
   $v.right = null$

The main difference from $\mathsf{Meld}(H, H')$ is Step 2 where the invariant I_1 is restored along the merged backbone. To establish the correctness of the implementation of $\mathsf{ColorMeld}(H, H')$ we consider each of the three steps in more details.

In Step 1 we merge the backbones of H and H' together such that the resulting tree is a heap-ordered tree that stores all elements in H and H' with unchanged keys. Since the merging does not change the left-children or the elements of any node and since H and H' both obey I_2 and I_3, the constructed heap-ordered tree also obeys I_2 and I_3. The merged backbone can however contain neighboring nodes of equal color. These conflicts are a violation of I_1.

In Step 2 we restore I_1. We solve all conflicts on the merged backbone between neighboring nodes v and w of equal color by letting the parent v swallow the child w as illustrated in Figure 6. We observe that since H and H' both obey I_1 a conflict must involve a node from both of them. This implies that a conflict can only occur in the part of the merged backbone made of nodes popped off the backbone-stacks in Step 1. We also observe that solving a conflict does not induce a new conflict. Combined with the previous observation this implies that the number of conflicts is bounded by the number of nodes popped off the backbone-stacks in Step 1. Finally, we observe that solving a conflict does not induce violations of I_2 and I_3, so after solving all conflicts on the merged backbone we have a colored heap-tree that stores all elements in H and H' with unchanged keys.

In Step 3 we shorten the merged backbone. It is done by moving the right-child of r' to its left-children, or in case r' has been swallowed by a node v in Step 2, by moving the right-child of v to its left-children. The subtree rooted by the right-child moved follows along, and thus becomes a subtree rooted by the new left-child of r' (or v). To argue that shortening the backbone does not induce violations of I_2 and I_3

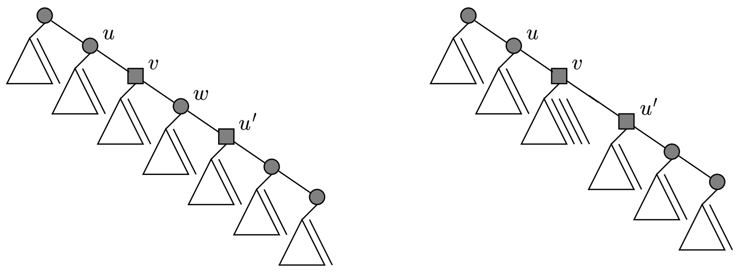

FIG. 6: This figure illustrates how a conflict on the merged backbone is solved. If $color(v) = color(w)$ then I_1 is violated. The invariant is restored by letting node v swallow node w, i.e. melding the elements and left-children at the two nodes and removing node w. Since $color(u) \neq color(w) = color(v)$ and $color(u') \neq color(v)$, solving a conflict does not induce another conflict.

we start by making two observations. First, we observe that moving the right-child of a node that obeys I_3 to its set of left-children results in a node that obeys I_2. Secondly, we observe that if a node that obeys I_2 (or I_3) swallows a node that obeys I_2 it results in a node that still obeys I_2 (or I_3).

Since r' is the root of H', it obeys I_3 before Step 2. We consider two cases. First, if r' is not swallowed in Step 2, the first observation immediately implies that it obeys I_2 after Step 3. Secondly, if r' is swallowed by a node v in Step 2, we might as well think of Steps 2 and 3 as occurring in opposite order as this does not affect the resulting tree. Hence, first we move the right-child of r' to its set of left-children, which by the first observation results in a node that obeys I_2, then we let node v swallow this node, which by the second observation does not affect the invariants obeyed by v.

We conclude that the implementation of $\mathsf{ColorMeld}(H, H')$ constructs a colored heap-tree that obeys all three invariants and stores all elements in H and H' with unchanged keys and colors. The backbone-stack of the colored heap-tree constructed by $\mathsf{ColorMeld}(H, H')$ is what remains on the backbone-stack of H after popping nodes in Step 1 with the node r' pushed onto it, unless the node r' is swallowed in Step 2.

Now consider the time it takes to meld n colored heap-trees of size one together by a sequence of $n-1$ melds. If we ignore the time it takes to meld the heap-trees storing elements and references to left-children when solving conflicts in Step 2 and shortening the backbone in Step 3, then we can bound the time it takes to do the sequence of melds by $O(n)$ exactly as we did in the previous section. Melding n colored heap-trees of size one involves melding at most n heap-trees of size one storing elements, and at most n heap-trees of size one storing references to left-children. Since melding n heap-trees of size one takes time $O(n)$, we have that melding the heap-trees storing elements and references to left-children also takes time $O(n)$, so melding n colored heap-trees of size one takes time $O(n)$ in the worst case.

4.2 Algorithms

In the following we present two algorithms to find pairs with lower bounded gap. First we describe a simple algorithm to find all right-maximal pairs with lower bounded gap using heap-trees, then we extend it to find all maximal pairs with lower bounded gap using colored heap-trees. Both algorithms run in time $O(n+z)$ where z is the number of reported pairs.

4.2.1 Right-maximal pairs with lower bounded gap

We find all right-maximal pairs in S with gap at least $g(|\alpha|)$, by for each node v in the binary suffix tree $T_B(S)$ considering the leaf-lists at its two children w_1 and w_2. The pair $(p, q, |\alpha|)$, for $p \in LL(w_1)$ and $q \in LL(w_2)$, is right-maximal and has gap at least $g(|\alpha|)$ if and only if $q \geq p + |\alpha| + g(|\alpha|)$. If we let p_{min} denote the minimum element in $LL(w_1)$ this implies that every q in

$$Q = \{q \in LL(w_2) \mid q \geq p_{min} + |\alpha| + g(|\alpha|)\}$$

forms a right-maximal pair $(p, q, |\alpha)$ with gap at least $g(|\alpha|)$ with every p in

$$P_q = \{p \in LL(w_1) \mid p \leq q - g(|\alpha|) - |\alpha|\} .$$

By construction P_q contains p_{min} and we have that $(p, q, |\alpha|)$ is a right-maximal pair with gap at least $g(|\alpha|)$ if and only if $q \in Q$ and $p \in P_q$. We can construct Q and P_q using heap-trees. Let H_i and $\bar{H}_i$ be heap-trees that store the elements in $LL(w_i)$ ordered by "$\leq$" and "$\geq$" respectively. By definition of the operations Min and Find we have that $p_{min} = \mathsf{Min}(H_1)$, $Q = \mathsf{Find}(\bar{H}_2, p_{min} + |\alpha| + g(|\alpha|)$ and $P_q = \mathsf{Find}(H_1, q - g(|\alpha|) - |\alpha|)$.

This leads to the formulation of Algorithm 3 in which we at every node v in $T_B(S)$ construct two heap-trees, H and $\bar{H}$, that store the elements in $LL(v)$ ordered by "$\leq$" and "$\geq$" respectively. If v is a leaf, we construct H and $\bar{H}$ directly by creating two heap-trees of size one each storing the index at the leaf. If v is an internal node, we construct H and $\bar{H}$ by melding the corresponding heap-trees at the two children (lines 11–12). Before constructing H and $\bar{H}$ at node v, we report right-maximal pairs of its path-label (lines 1–10).

To argue that Algorithm 3 finds all right-maximal pairs in S with gap at least $g(|\alpha|)$ it is sufficient to show that we at each node v in $T_B(S)$ report all pairs $(p, q, |\alpha|)$ and $(q, p, |\alpha|)$, where $p \in LL(w_1)$ and $q \in LL(w_2)$, with gap at least $g(|\alpha|)$. The rest follows because we consider every node in $T_B(S)$. Let v be a node in $T_B(S)$ at which the heap-trees H_1, $\bar{H}_1$, H_2, and $\bar{H}_2$ at its two children are available. As explained above $(p, q, |\alpha|)$ is a right-maximal pair with gap at least $g(|\alpha|)$ if and only if $q \in Q$ and $p \in P_q$, which are exactly the pairs reported in lines 1–5. Symmetrically we can argue that $(q, p, |\alpha|)$ is a right-maximal pair with gap at least $g(|\alpha|)$ if and only if $p \in P$ and $q \in Q_p$, which are exactly the pairs reported in lines 6–10.

Now consider the running time of the algorithm. We first note that constructing two heap-trees of size one at each of the n leaves in $T_B(S)$ and melding them together according to the structure of $T_B(S)$ takes time $O(n)$ because each of the $n-1$ meld

Algorithm 3 Find all right-maximal pairs in S with lower bounded gap.

1. *Initializing:* Build the binary suffix tree $T_B(S)$. Create at each leaf two heap-trees of size one, H ordered by "$\leq$" and $\bar{H}$ ordered by "$\geq$", that both store the index at the leaf.
2. *Reporting and melding:* When the heap-trees H_1 and $\bar{H}_1$ at the left-child of node v, and the heap-trees H_2 and $\bar{H}_2$ at the right-child of node v are available we report pairs of α, the path-label of v, and construct the heap-trees H and $\bar{H}$ as follows

```
1   $Q = \mathsf{Find}(\bar{H}_2, \mathsf{Min}(H_1) + |\alpha| + g(|\alpha|))$
2   for $q$ in $Q$ do
3       $P_q = \mathsf{Find}(H_1, q - g(|\alpha|) - |\alpha|)$
4       for $p$ in $P_q$ do
5           report pair $(p, q, |\alpha|)$

6   $P = \mathsf{Find}(\bar{H}_1, \mathsf{Min}(H_2) + |\alpha| + g(|\alpha|))$
7   for $p$ in $P$ do
8       $Q_p = \mathsf{Find}(H_2, p - g(|\alpha|) - |\alpha|)$
9       for $q$ in $Q_p$ do
10          report pair $(q, p, |\alpha|)$

11  $H = \mathsf{Meld}(H_1, H_2)$
12  $\bar{H} = \mathsf{Meld}(\bar{H}_1, \bar{H}_2)$
```

operation takes amortized constant time. We then note that the reporting of pairs at each node, lines 1–10, takes time proportional to the number of reported pairs because the find operation takes time proportional to the number of returned elements and the set P_q (and Q_p) is non-empty for every element q in Q (and p in P). Finally we recall that constructing the binary suffix tree $T_B(S)$ takes time $O(n)$. Now consider the space needed by the algorithm. The binary suffix tree requires space $O(n)$. The heap-trees also requires space $O(n)$ because no element at any time is stored in more than one heap-tree. Finally, since no leaf-list contains more than n elements, storing the elements returned by the find operations during the reporting requires no more than space $O(n)$. In summary we formulate the following theorem.

THEOREM 4.1
Algorithm 3 finds all right-maximal pairs $(i, j, |\alpha|)$ in a string S of length n with gap at least $g(|\alpha|)$ in space $O(n)$ and time $O(n + z)$, where z is the number of reported pairs.

4.2.2 Maximal pairs with lower bounded gap

Essential to the above algorithm is that we in time proportional to its size can construct the set Q that contains all elements q in $LL(w_2)$ that form a right-maximal pair $(p_{min}, q, |\alpha|)$ with gap at least $g(|\alpha|)$. Unfortunately the left-characters $S[q-1]$

and $S[p_{min} - 1]$ can be equal, so Q can contain elements that do not form a maximal pair with any element in $LL(w_1)$. Since we aim for the reporting of pairs to take time proportional to the number of reported pairs, this implies that we cannot afford to consider every element in Q if we only want to report maximal pairs.

Fortunately we can efficiently construct the subset of $LL(w_2)$ that contains all the elements that form at least one maximal pair. An element q in $LL(w_2)$ forms a maximal pair if and only if there is an element p in $LL(w_1)$ such that $q \geq p + |\alpha| + g(|\alpha|)$ and $S[q-1] \neq S[p-1]$. We can construct this subset of $LL(w_2)$ using colored heap-trees. We define the color of an element to be its left-character, i.e. the color of p in $LL(w_1)$ and q in $LL(w_2)$ is $S[p-1]$ and $S[q-1]$ respectively. Let H_i and $\bar{H}_i$ be colored heap-trees that store the elements in $LL(w_i)$ ordered by "$\leq$" and "$\geq$" respectively. Using $p_{min} = \mathsf{ColorMin}(H_1)$ and $p_{sec} = \mathsf{ColorSec}(H_1)$ we can characterize the elements in $LL(w_2)$ that form at least one maximal pair with gap at least $g(|\alpha|)$ by considering two cases.

First, if $q \geq p_{sec} + |\alpha| + g(|\alpha|)$ then $(p_{min}, q, |\alpha|)$ and $(p_{sec}, q, |\alpha|)$ both have gap at least $g(|\alpha|)$ and since $S[p_{min} - 1] \neq S[p_{sec} - 1]$ at least one of them is maximal, so every $q \geq p_{sec} + |\alpha| + g(|\alpha|)$ forms a maximal pair with gap at least $g(|\alpha|)$. If $\#$ is a character not appearing anywhere in S, i.e. no element in $LL(w_2)$ has color $\#$, this is the same as saying that every q in $Q' = \mathsf{ColorFind}(\bar{H}_2, p_{sec} + |\alpha| + g(|\alpha|), \#)$ forms a maximal pair with gap at least $g(|\alpha|)$. Secondly, if $q < p_{sec} + |\alpha| + g(|\alpha|)$ forms a maximal pair $(p, q, |\alpha|)$ with gap at least $g(|\alpha|)$ then $p_{min} \leq p < p_{sec}$. This implies that $S[p-1] = S[p_{min} - 1]$, so $(p_{min}, q, |\alpha|)$ is also maximal and has gap at least $g(|\alpha|)$. We thus have that $q < p_{sec} + |\alpha| + g(|\alpha|)$ forms a maximal pair with gap at least $g(|\alpha|)$ if and only if $(p_{min}, q, |\alpha|)$ is maximal and has gap at least $g(|\alpha|)$, i.e. if and only if $S[q-1] \neq S[p_{min} - 1]$ and $q \geq p_{min} + |\alpha| + g(|\alpha|)$. This implies that the set $Q'' = \mathsf{ColorFind}(\bar{H}_2, p_{min} + |\alpha| + g(|\alpha|), S[p_{min} - 1])$ contains every $q < p_{sec} + |\alpha| + g(|\alpha|)$ that forms a maximal pair with gap at least $g(|\alpha|)$.

By construction of Q' and Q'' the set $Q' \cup Q''$ contains all elements in $LL(w_2)$ that form a maximal pair with gap at least $g(|\alpha|)$. More precisely, every q in the set $Q' \cup Q''$ forms a maximal pair $(p, q, |\alpha|)$ with gap at least $g(|\alpha|)$ with every $p \leq q - g(|\alpha|) - |\alpha|$ in $LL(w_1)$ where $S[p-1] \neq S[q-1]$, i.e. with every p in the set $P_q = \mathsf{ColorFind}(H_1, q - g(|\alpha|) - |\alpha|, S[q-1])$ which by construction is non-empty. We can construct the set $Q' \cup Q''$ efficiently as follows. Every element in Q'' greater than $p_{sec} + |\alpha| + g(|\alpha|)$ is also in Q', so we can construct $Q' \cup Q''$ by concatenating Q' and what remains of Q'' after removing all elements greater than $p_{sec} + |\alpha| + g(|\alpha|)$ from it. Combined with the complexity of $\mathsf{ColorFind}$ this implies that we can construct the set $Q' \cup Q''$ in time proportional to $|Q'| + |Q''| \leq 2|Q' \cup Q''|$.

This leads to the formulation of Algorithm 4. The algorithm is similar to Algorithm 3 except that we maintain colored heap-trees during the traversal of the binary suffix tree. At every node we report maximal pairs of its path-label. In lines 1–7 we report all maximal pairs $(p, q, |\alpha|)$ by constructing and considering the elements in P_q for every q in $Q' \cup Q''$. In lines 8–15 we analogously report all maximal pairs $(q, p, |\alpha|)$. The correctness of the algorithm follows immediately from the above discussion. Since the operations on colored heap-trees have the same complexities as the corresponding operations on heap-tress, the running time and space requirement

Algorithm 4 Find all maximal pairs in S with lower bounded gap.

1. *Initializing:* Build the binary suffix tree $T_B(S)$. Create at each leaf two colored heap-trees of size one, H ordered by "$\leq$" and $\bar{H}$ ordered by "$\geq$", that both store the index at the leaf with color corresponding to its left-character.
2. *Reporting and melding:* When the colored heap-trees H_1 and $\bar{H}_1$ at the left-child of node v, and the colored heap-trees H_2 and $\bar{H}_2$ at the right-child of node v are available we report pairs of α, the path-label of v, and construct the colored heap-trees H and $\bar{H}$ as follows, where $\#$ is a character not appearing anywhere in S.

```
1   $p_{min}, p_{sec} = \mathsf{ColorMin}(H_1), \mathsf{ColorSec}(H_1)$
2   $Q' = \mathsf{ColorFind}(\bar{H}_2, p_{sec} + |\alpha| + g(|\alpha|), \#)$
3   $Q'' = \mathsf{ColorFind}(\bar{H}_2, p_{min} + |\alpha| + g(|\alpha|), S[p_{min} - 1])$
4   for $q$ in $Q' \cup Q''$ do
5       $P_q = \mathsf{ColorFind}(H_1, q - g(|\alpha|) - |\alpha|, S[q - 1])$
6       for $p$ in $P_q$ do
7           report pair $(p, q, |\alpha|)$

8   $q_{min}, q_{sec} = \mathsf{ColorMin}(H_2), \mathsf{ColorSec}(H_2)$
9   $P' = \mathsf{ColorFind}(\bar{H}_1, q_{sec} + |\alpha| + g(|\alpha|), \#)$
10  $P'' = \mathsf{ColorFind}(\bar{H}_1, q_{min} + |\alpha| + g(|\alpha|), S[q_{min} - 1])$
11  for $p$ in $P' \cup P''$ do
12      $Q_p = \mathsf{ColorFind}(H_2, p - g(|\alpha|) - |\alpha|, S[p - 1])$
13      for $q$ in $Q_p$ do
14          report pair $(q, p, |\alpha|)$

15  $H = \mathsf{ColorMeld}(H_1, H_2)$
16  $\bar{H} = \mathsf{ColorMeld}(\bar{H}_1, \bar{H}_2)$
```

of the algorithm is exactly as analyzed for Algorithm 3. In summary we can formulate the following theorem.

THEOREM 4.2
Algorithm 4 finds all maximal pairs $(i, j, |\alpha|)$ in a string S of length n with gap at least $g(|\alpha|)$ in space $O(n)$ and time $O(n + z)$, where z is the number of reported pairs.

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 [25]. 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 Nørgaard Storm Pedersen and Jens Stoye were visiting Dan Gusfield at UC Davis. We would like to thank Dan Gusfield and Rob Irwing for listening to some early 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. In *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1645 of *Lecture Notes in Computer Science*, pages 134–149, 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. Transducers 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] R. W. Floyd. Algorithm 245: Treesort3. *Communications of the ACM*, 7(12):701, 1964.

[9] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 338–346, 1984.

[10] 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.

[11] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[12] 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.

[13] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[14] 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.

[15] S. Karlin, M. Morris, G. Ghandour, and M.-Y. Leung. Efficient algorithms for molecular sequence analysis. *Proceedings of the National Academy of Science of the USA*, 85:841–845, 1988.

[16] R. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 596–604, 1999.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[23] K. Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1994.

[24] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[25] 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.

[26] 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.

[27] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6:306–318, 1985.

[28] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

[29] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[30] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[31] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

Received May 8, 2000.