Alcom-FT Technical Report Series
ALCOMFT-TR-02-55

Solving the String Statistics Problem in Time $\mathcal{O}(n \log n)$

Gerth Stølting Brodal[1,⋆,⋆⋆], Rune B. Lyngsø[3],
Anna Östlin[1,⋆⋆], and Christian N. S. Pedersen[1,2,⋆⋆]

[1] BRICS[⋆⋆⋆], Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Århus C, Denmark. E-mail: {gerth,annao,cstorm}@brics.dk
[2] BiRC[†], University of Aarhus, Ny Munkegade, DK-8000 Århus C, Denmark.
[3] Department of Statistics, Oxford University, Oxford OX1 3TG, UK.
E-mail: lyngsoe@stats.ox.ac.uk

Abstract The string statistics problem consists of preprocessing a string of length n such that given a query pattern of length m, the maximum number of non-overlapping occurrences of the query pattern in the string can be reported efficiently. Apostolico and Preparata introduced the minimal augmented suffix tree (MAST) as a data structure for the string statistics problem, and showed how to construct the MAST in time $\mathcal{O}(n \log^2 n)$ and how it supports queries in time $\mathcal{O}(m)$ for constant sized alphabets. A subsequent theorem by Fraenkel and Simpson stating that a string has at most a linear number of distinct squares implies that the MAST requires space $\mathcal{O}(n)$. In this paper we improve the construction time for the MAST to $\mathcal{O}(n \log n)$ by extending the algorithm of Apostolico and Preparata to exploit properties of efficient joining and splitting of search trees together with a refined analysis.

1 Introduction

The *string statistics problem* consists of preprocessing a string S of length n such that given a query pattern α of length m, the maximum number of non-overlapping occurrences of α in S can be reported efficiently. Without preprocessing the maximum number of non-overlapping occurrences of α in S can be found in time $\mathcal{O}(n)$, by using a linear time string matching algorithm to find all occurrences of α in S, e.g. the algorithm by Knuth, Morris, and Pratt [14], and then in a greedy fashion from left-to-right compute the maximal number of non-overlapping occurrences.

⋆ Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).
⋆⋆ Supported by the Carlsberg Foundation (contract number ANS-0257/20).
⋆⋆⋆ Basic Research in Computer Science (BRICS), www.brics.dk, funded by the Danish National Research Foundation.
† Bioinformatics Research Center (BiRC), www.birc.dk, funded by Aarhus University Research Fundation.

Apostolico and Preparata in [3] described a data structure for the string statistics problem, *the minimal augmented suffix tree* $\mathsf{MAST}(S)$, with preprocessing time $\mathcal{O}(n \log^2 n)$ and with query time $\mathcal{O}(m)$ for constant sized alphabets. In this paper we present an improved algorithm for constructing $\mathsf{MAST}(S)$ with preprocessing time $\mathcal{O}(n \log n)$, and prove that $\mathsf{MAST}(S)$ requires space $\mathcal{O}(n)$, which follows from a recent theorem of Fraenkel and Simpson [9].

The basic idea of the algorithm of Apostolico and Preparata and our algorithm for constructing $\mathsf{MAST}(S)$, is to perform a traversal of the suffix tree of S while maintaining the leaf-lists of the nodes visited in appropriate data structures (see Section 1.1 for definition details). Traversing the suffix tree of a string to construct and examine the leaf-lists at each node is a general technique for finding regularities in a string, e.g. for finding squares in a string (or tandem repeats) [2,17], for finding maximal quasi-periodic substrings, i.e. substrings that can be covered by a shorter substring, [1,6], and for finding maximal pairs with bounded gap [4]. All these problems can be solved using this technique in time $\mathcal{O}(n \log n)$. Other applications are listed by Gusfield in [10, Chapter 7].

A crucial component of our algorithm is the representation of a leaf list by a collection of search trees, such that the leaf-list of a node in the suffix tree of S can be constructed from the leaf-lists of the children by efficient merging. Hwang and Lin [13] described how to optimally merge two sorted lists of length n_1 and n_2, where $n_1 \leq n_2$, with $\mathcal{O}(n_1 \log \frac{n_1+n_2}{n_1})$ comparisons. Brown and Tarjan [7] described how to achieve the same number of comparisons for merging two AVL-trees in time $\mathcal{O}(n_1 \log \frac{n_1+n_2}{n_1})$, and Huddleston and Mehlhorn [12] showed a similar result for level-linked (2,4)-trees. In our algorithm we will use a slightly extended version of level-linked (2,4)-trees where each element has an associated weight. Due to lack of space proofs have been omitted. The omitted details can be found in [5].

1.1 Preliminaries

Some of the terminology and notation used in the following originates from [3], but with minor modifications. We let Σ denote a finite alphabet, and for a string $S \in \Sigma^*$ we let $|S|$ denote the length of S, $S[i]$ the ith character in S, for $1 \leq i \leq |S|$, and $S[i\,..\,j] = S[i]S[i+1]\cdots S[j]$ the substring of S from the ith to the jth character, for $1 \leq i \leq j \leq |S|$. The suffix $S[i\,..\,|S|]$ of S starting at position i will be denoted $S[i\,..\,]$.

An integer p, for $1 \leq p \leq |S|$, is denoted *a period* of S if and only if the suffix $S[p+1\,..\,]$ of S is also a prefix of S, i.e. $S[p+1\,..\,] = S[1\,..\,|S|-p]$. The shortest period p of S is denoted *the period* of S, and the string S is said to be *periodic* if and only if $p \leq |S|/2$. A nonempty string S is a *square*, if $S = \alpha\alpha$ for some string α.

In the rest of this paper S denotes the input string with length n and α a substring of S. A non-empty string α is said to *occur* in S at position i if $\alpha = S[i\,..\,i+|\alpha|-1]$ and $1 \leq i \leq n-|\alpha|+1$. E.g. in the string $\underline{b\,a\,b}\,a\,a\,a\,a\,\underline{b\,a\,b}\,a\,a\,b$ the substring $b\,a\,b$ occurs at positions 1 and 8. The *maximum number of non-overlapping occurrences* of a string α in a string S, is the maximum number of occurrences of α where no two occurrences overlap. E.g. the maximum number of non-overlapping occurrences of $b\,a\,b$ in $b\,a\,b\,a\,b\,a\,b\,a\,b\,a\,b$ is three, since the occurrences at positions 1, 5 and 9 do not overlap.

The *suffix tree* $\mathsf{ST}(S)$ of the string S is the compressed trie storing all suffixes of the string $S\$$ where $\$ \notin \Sigma$. Each leaf in $\mathsf{ST}(S)$ represents a suffix $S[i\,..]\$$ of $S\$$ and is annotated with the index i. Each edge in $\mathsf{ST}(S)$ is labeled with a nonempty substring of $S\$$, represented by the start and end positions in S, such that the path from the root to the leaf annotated with index i spells the suffix $S[i\,..]\$$. We refer to the substring of S spelled by the path from the root to a node v as the *path-label* of v and denote it $\mathsf{L}(v)$. We refer to the set of indices stored at the leaves of the subtree rooted at v as the *leaf-list* of v and denote it $\mathsf{LL}(v)$. Since $\mathsf{LL}(v)$ is exactly the set of start positions i where $\mathsf{L}(v)$ is a prefix of the suffix $S[i\,..]\$$, we have Fact 1 below.

Fact 1 *If v is an internal node of $\mathsf{ST}(S)$, then $\mathsf{LL}(v) = \bigcup_{c \text{ child of } v} \mathsf{LL}(c)$, and $i \in \mathsf{LL}(v)$ if and only if $\mathsf{L}(v)$ occurs at position i in S.*

The problem of constructing $\mathsf{ST}(S)$ has been studied intensively and several algorithms have been developed which for constant sized alphabets can construct $\mathsf{ST}(S)$ in time and space $\mathcal{O}(|S|)$ [8,15,18,19]. For non-constant alphabet sizes the running time of the algorithms become $\mathcal{O}(|S|\log|\Sigma|)$.

In the following we let the height of a tree T be denoted $h(T)$ and be defined as the maximum number of edges in a root-to-leaf path in T, and let the size of T be denoted $|T|$ and be defined as the number of leaves of T. For a node v in T we let T_v denote the subtree of T rooted at node v, and let $|v| = |T_v|$ and $h(v) = h(T_v)$. Finally, for a node v in a binary tree we let $\mathrm{small}(v)$ denote the child of v with smaller size (ties are broken arbitrarily).

The basic idea of our algorithm in Section 5 is to process the suffix tree of the input string bottom-up, such that we at each node v spend amortized time $\mathcal{O}(|\mathrm{small}(v)| \cdot \log(|v|/|\mathrm{small}(v)|))$. Lemma 1 then states that the total time becomes $\mathcal{O}(n \log n)$ [16, Exercise 35].

Lemma 1. *Let T be a binary tree with n leaves. If for every internal node v, $c_v = |\mathrm{small}(v)| \cdot \log(|v|/|\mathrm{small}(v)|)$, and for every leaf v, $c_v = 0$, then $\sum_{v \in T} c_v \leq n \log n$.*

2 The String Statistics Problem

Given a string S of length n and a pattern α of length m the following greedy algorithm will compute the maximum number of non-overlapping occurrences of α in S. Find all occurrences of α in S by using an exact string matching algorithm. Choose the leftmost occurrence. Continue to choose greedily the leftmost occurrence not overlapping with any so far chosen occurrence. This greedy algorithm will compute the maximum number of occurrences of α in S in time $\mathcal{O}(n)$, since all matchings can be found in time $\mathcal{O}(n)$, e.g. by the algorithm by Knuth, Morris, and Pratt [14].

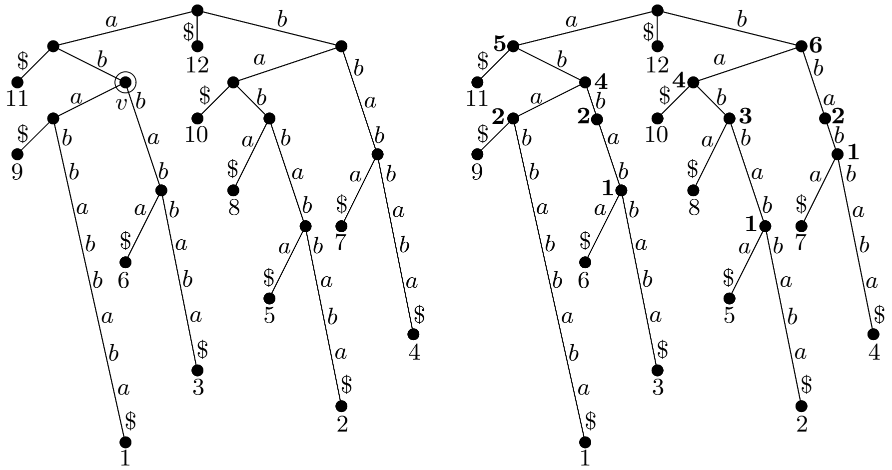

Figure 1. To the left is the suffix tree $\mathsf{ST}(S)$ of the string $S = a\,b\,a\,b\,b\,a\,b\,b\,a\,b\,a$. The node v has path-label $\mathsf{L}(v) = a\,b$ and leaf-list $\mathsf{LL}(v) = \{1, 3, 6, 9\}$. To the right is the minimal augmented suffix tree $\mathsf{MAST}(S)$ for the string $S = a\,b\,a\,b\,b\,a\,b\,b\,a\,b\,a$. Numbers in the internal nodes are the c-values.

In the *string statistics problem* we want to preprocess a string S such that queries of the following form are supported efficiently: Given a query string α, what is the maximum number of non-overlapping occurrences of α in S? The maximum number of non-overlapping occurrences of α is called the *c-value* of α, denoted $c(\alpha)$. The preprocessing will be to compute the *minimal augmented suffix tree* described below. Given the minimal augmented suffix tree, string statistics queries can be answered in time $\mathcal{O}(m)$.

For any substring, α, of S there is exactly one path from the root of $\mathsf{ST}(S)$ ending in a node or on an edge of $\mathsf{ST}(S)$ spelling out the string α. This node or edge is called the *locus* of α. In a suffix tree $\mathsf{ST}(S)$ the number of leaves in the subtree below the locus of α in $\mathsf{ST}(S)$ tells us the number of occurrences of α in S. These occurrences may overlap, hence the suffix tree is not immediately suitable for the string statistics problem. The minimal augmented suffix tree for S, denoted $\mathsf{MAST}(S)$ can be constructed from the suffix tree $\mathsf{ST}(S)$ as follows. A minimum number of new auxiliary nodes are inserted into $\mathsf{ST}(S)$ in such a way that the c-value for all substrings with locus on an edge (u, v), where u is the parent of v, have c-value equal to $c(\mathsf{L}(v))$, i.e. the c-value only changes at internal nodes along a path from a leaf to the root. Each internal node v in the augmented tree is then labeled by $c(\mathsf{L}(v))$ to get the minimal augmented suffix tree. Figure 1 shows the suffix tree and the minimal augmented suffix tree for the string *ababbabbaba*.

Fraenkel and Simpson in [9] prove that a string S contains less than $2|S|$ distinct squares, which implies the following lemma.

Lemma 2. *The minimal augmented suffix tree for a string S has at most $3|S|$ internal nodes.*

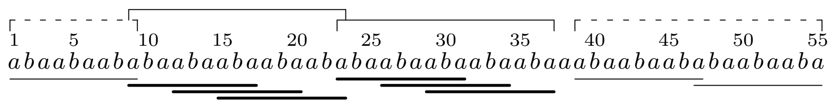

Figure 2. The grouping of occurrences in a string into chunks and necklaces. Occurrences are shown below the string. Thick lines are occurrences in chunks. The grouping into chunks and necklaces is shown above the string. Necklaces are shown using dashed lines. Note that a necklace can consist of a single occurrence.

It follows that the space needed to store $\mathsf{MAST}(S)$ is $\mathcal{O}(n)$.

3 String Properties

The lemma below gives a characterization of how the occurrences of a string α can appear in S (proof omitted).

Lemma 3. *Let S be a string and α a substring of S. If the occurrences of α in S are at positions $i_1 < \cdots < i_k$, then for all $1 \leq j < k$ either $i_{j+1} - i_j = p$ or $i_{j+1} - i_j > \max\{|\alpha| - p, p\}$, where p denotes the period of α.*

A consequence of Lemma 3 is that if $p \geq |\alpha|/2$, then an occurrence of α in S at position i_j can only overlap with the occurrences at positions i_{j-1} and i_{j+1}. If $p < |\alpha|/2$, then two consecutive occurrences i_j and i_{j+1}, either satisfy $i_{j+1} - i_j = p$ or $i_{j+1} - i_j > |\alpha| - p$.

Corollary 1. *If $i_{j+1} - i_j \leq |\alpha|/2$, then $i_{j+1} - i_j = p$ where p is the period of α.*

Motivated by the above observations we group the occurrences of α in S into *chunks* and *necklaces*. Let p denote the period of α. Chunks can only appear if $p < |\alpha|/2$. A chunk is a maximal sequence of occurrences containing at least two occurrences and where all consecutive occurrences have distance p. The remaining occurrences are grouped into necklaces. A necklace is a maximal sequence of overlapping occurrences, i.e. only two consecutive occurrences overlap at a given position and the overlap of two occurrences is between one and $p - 1$ positions long. Figure 2 shows the occurrences of the string *abaabaaba* in a string of length 55 grouped into chunks and necklaces. By definition two necklaces cannot overlap, but a chunk can overlap with another chunk or a necklace at both ends. By Lemma 3 the overlap is at most $p - 1$ positions.

We now turn to the contribution of chunks and necklaces to the c-values. We first consider the case where chunks and necklaces do not overlap. An *isolated* necklace or chunk is a necklace or chunk that does not overlap with other necklaces and chunks. Figure 3 gives an example of the contribution to the c-values by an isolated necklace and chunk. More formally, we have the following lemma, which we state without proof.

Lemma 4. *An isolated necklace of k occurrences of α contributes to the c-value of α with $\lceil k/2 \rceil$. An isolated chunk of k occurrences of α contributes to the c-value of α with $\lceil k/\lceil |\alpha|/p \rceil \rceil$, where p is the period of α.*

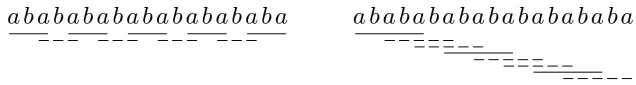

Figure 3. Examples of the contribution to the c-values by an isolated necklace (left; $\alpha = aba$ and the contribution is $5 = \lceil 9/2 \rceil$) and an isolated chunk (right; $\alpha = ababa$, $p = 2$, and the contribution is $3 = \lceil 8/\lceil 5/2 \rceil \rceil$)

Motivated by Lemma 4, we define the *nominal contribution* of a necklace of k occurrences of α to be $\lceil k/2 \rceil$ and the nominal contribution of a chunk of k occurrences of α to be $\lceil k/\lceil |\alpha|/p \rceil \rceil$. The nominal contribution of a necklace or chunk of α's is the contribution to the c-value of α if the necklace or chunk appeares isolated. If the necklace of chunk does not appear isolated, i.e. it overlaps with a neighboring necklace or chunk, then its actual contribution to the c-value of α is at most be one less than its nominal contribution to the c-value of α.

We define the *excess* of a necklace of k occurrences to be $(k-1) \bmod 2$, and the excess of a chunk of k occurrences to be $(k-1) \bmod \lceil |\alpha|/p \rceil$. The excess describes the number of occurrences of $\alpha[1 \mathinner{..} p]$ which are covered by the necklace or chunk, but not covered by the maximal sequence of non-overlapping occurences.

We group the chunks and necklaces into a collection of *chains* $\mathcal{C}$ by the following two rules:

1. A chunk with excess at least two is a chain by itself.
2. A maximal sequence of overlapping necklaces and chunks with excess zero or one is a chain.

For a chain $c \in \mathcal{C}$ we define $\#_0(c)$ to be the number of chunks and necklaces with excess zero in the chain.

We are now ready to state our main lemma enabling the efficient computation of the c-values. The lemma gives an alternative to the characterization in [3, Proposition 2] (proof omitted).

Lemma 5. *The maximum number of non-overlapping occurrences of α in S equals the sum of the nominal contributions of all necklaces and chunks minus* $\sum_{c \in \mathcal{C}} \lfloor \#_0(c)/2 \rfloor$.

4 Level-Linked (2,4)-Trees

In this section we consider how to maintain a set of sorted lists of elements as a collection of level-linked (2,4)-trees where the elements are stored at the leaves in sorted order from left-to-right, and each element can have an associated real valued weight. For a detailed treatment of level-linked (2,4)-trees see [12] and [16, Section III.5]. The operations we consider supported are:

$\mathsf{NewTree}(e, w)$: Creates a new tree T containing the element e with associated weight w.

$\mathsf{Search}(p, e)$: Search for the element e starting the search at the leaf of a tree T that p points to. Returns a reference to the leaf in T containing e or the immediate predecessor or successor of e.

$\mathsf{Insert}(p, e, w)$: Creates a new leaf containing the element e with associated weight w and inserts the new leaf immediate next to the leaf pointed to by p in a tree T, provided that the sorted order is maintained.

$\mathsf{Delete}(p)$: Deletes the leaf and element that p is a pointer to in a tree T.

$\mathsf{Join}(T_1, T_2)$: Concatenates two trees T_1 and T_2 and returns a reference to the resulting tree. It is required that all elements in T_1 are smaller than the elements in T_2 w.r.t. the total order.

$\mathsf{Split}(T, e)$: Splits the tree T into two trees T_1 and T_2, such that e is larger than all elements in T_1 and smaller than or equal to all elements in T_2. Returns references to the two trees T_1 and T_2.

$\mathsf{Weight}(T)$: Returns the sum of the weights of the elements in the tree T.

Theorem 1 (Hoffmann et al. [11, Section 3]). *Level-linked (2,4)-trees support* NewTree*,* Insert *and* Delete *in amortized constant time,* Search *in time* $\mathcal{O}(\log d)$ *where* d *is the number of elements in* T *between* e *and* p*, and* Join *and* Split *in amortized time* $\mathcal{O}(\log \min\{|T_1|, |T_2|\})$.

To allow each element to have an associated weight we extend the construction from [11, Section 3] such that we for all nodes v in a tree store the sum of the weights of the leaves in the subtree T_v, except for the nodes on the paths to the leftmost and rightmost leaves. These sums are straightforward to maintain while rebalancing a (2,4)-tree under node splittings and fusions, since the sum at a node is the sum of the weights at the children of the node. For each tree we also store the total weight of the tree.

Theorem 2. *Weighted level-linked (2,4)-trees support* NewTree *and* Weight *in amortized constant time,* Insert *and* Delete *in amortized time* $\mathcal{O}(\log |T|)$*,* Search *in time* $\mathcal{O}(\log d)$ *where* d *is the number of elements in* T *between* e *and* p*, and* Join *and* Split *in amortized time* $\mathcal{O}(\log \min\{|T_1|, |T_2|\})$.

5 The Algorithm

In this section we describe the algorithm for constructing the minimal augmented suffix tree for a string S of length n.

Algorithm idea: The algorithm starts by constructing the suffix tree, $\mathsf{ST}(S)$, for S. The suffix tree is then augmented with extra nodes and c-values for all nodes to get the minimal augmented suffix tree, $\mathsf{MAST}(S)$, for S. The augmentation of $\mathsf{ST}(S)$ to $\mathsf{MAST}(S)$ starts at the leaves and the tree is processed in a bottom-up fashion. At each node v encountered on the way up the tree the c-value for the path-label $\mathsf{L}(v)$ is added to the tree, and at each edge new nodes and their c-values are added if there is a change in the c-value along the edge. To be able to efficiently compute the c-values and decide if new nodes should be added along edges the indices in the leaf-list of v, $\mathsf{LL}(v)$, are stored in a data structure that keeps track of necklaces, chunks, and chains, as defined in Section 3.

Data structure: Let α be a substring of S. The data structure $\mathsf{D}(\alpha)$ is a search tree for the indices of the occurrences of α in S. The leaves in $\mathsf{D}(\alpha)$ are the leaves in $\mathsf{LL}(v)$, where v is the node in $\mathsf{ST}(S)$ such that the locus of α is the edge directly above v or the node v. The search tree, $\mathsf{D}(\alpha)$, will be organized into three levels to keep track of chains, chunks, and necklaces. The top level in the search tree stores chains, the middle level chunks and necklaces, and the bottom level occurrences.

Top level: Unweighted (2,4)-tree (cf. Theorem 1) with the chains as leaves. The leftmost indices in each chain are the keys.

Middle level: One weighted (2,4)-tree (cf. Theorem 2) for each chain, with the chunks and necklaces as leaves. The leftmost indices in each chunk or necklace are the keys. The weight of a leaf is 1 if the excess of the chunk or necklace is zero, otherwise the weight is 0. The total weight of a tree on the middle level is $\#_0(c)$, where c denotes the chain represented by the tree.

Bottom level: One weighted (2,4)-tree for each chunk and necklace, with the occurrences in the chunk or necklace as the leaves. The weight of a leaf is one. The total weight of a tree is the number of occurrences in the chunk or the necklace.

Together with each of the 3-level search trees, $\mathsf{D}(\alpha)$, some variables are stored. $\mathsf{NCS}(\alpha)$ stores the sum of the nominal contribution for all chunks and necklaces, $\mathsf{ZS}(\alpha)$ stores the sum $\sum_{c\in\mathcal{C}}\lceil\#_0(c)/2\rceil$, where $\mathcal{C}$ is the set of chains. By Lemma 5 the maximum number of non-overlapping occurrences of α is $\mathsf{NCS}(\alpha) - \mathsf{ZS}(\alpha)$. We also store the total number of indices in $\mathsf{D}(\alpha)$ and a list of all chunks denoted $\mathsf{CHUNKLIST}(\alpha)$. Finally we store, $p(\alpha)$, which is the smallest difference between the indices of two consecutive occurrences in $\mathsf{D}(\alpha)$. Note that, by Corollary 1, $p(\alpha)$ is the period of α if there is at least one chunk. To make our presentation more readable we will sometimes refer to the tree for a chain, chunk, or necklace just as the chain, chunk, or necklace.

For the top level tree in $\mathsf{D}(\alpha)$ we will use level-linked (2,4)-trees, according to Theorem 1, and for the middle and bottom level trees in $\mathsf{D}(\alpha)$ we will use weighted level-linked (2,4)-trees, according to Theorem 2. In these trees predecessor and successor queries are supported in constant time. We denote by $\ell(e)$ and $r(e)$ the indices to the left and right of index e. To be able to check fast if there are overlaps between two consecutive trees on the middle and bottom levels we store the first and last index in each tree in the root of the tree. This can easily be kept updated when the trees are joined and split.

We will now describe how the suffix tree is processed and how the data structures are maintained during this process.

Processing events: We want to process edges in the tree bottom-up, i.e. for decreasing length of α, so that new nodes are inserted if the c-value changes along the edge, the c-values for nodes are added to the tree, and the data structure is kept updated. The following events can cause changes in the c-value and the chain, chunk, and necklace structure.

1. Excess change: When $|\alpha|$ becomes $i \cdot p(\alpha)$, for $i = 2, 3, 4, \ldots$ the excess and nominal contribution of chunks changes and we have to update the data structure and possibly add a node to the suffix tree.
2. Chunks become necklaces: When $|\alpha|$ decreases and becomes $2p$ a chunk degenerates into a necklace. At this point we join all overlapping chunks and necklaces into one necklace and possibly add a node to the suffix tree.
3. Necklace and chain break-up: When $|\alpha|$ decreases two consecutive occurrences at some point no longer overlap. The result is that a necklace or a chain may split, and we have to update the necklace and chain structure and possibly add a node to the suffix tree.
4. Merging at internal nodes: At internal nodes in the tree the data structures for the subtrees below the node are merged into one data structure and the c-value for the node is added to the tree.

To keep track of the events we use an event queue, denoted EQ, that is a common priority queue of events for the whole suffix tree. The priority of an event in EQ is equal to the length of the string α when the event has to be processed. Events of type 1 and 2 store a pointer to any leaf in $\mathsf{D}(\alpha)$. Events of type 3, i.e. that two consecutive overlapping occurrences with index e_1 and e_2, $e_1 < e_2$, terminate to overlap, store a pointer to the leaf e_1 in the suffix tree. For the leaf e_1 in the suffix tree also a pointer to the event in EQ is stored. Events of type 4 stores a pointer to the internal node in the suffix tree involved in the event. When the suffix tree is constructed all events of type 4 are inserted into EQ. For a node v in $\mathsf{ST}(S)$ the event has priority $|\mathsf{L}(v)|$ and stores a pointer to v. The pointers are used to be able to decide which data structure to update. The priority queue EQ is implemented as a table with entries $\mathsf{EQ}[1] \ldots \mathsf{EQ}[|S|]$. All events with priority x are stored in a linked list in entry $\mathsf{EQ}[x]$. Since the priorities of the events considered are monotonic decreasing, it is sufficient to consider the entries of EQ in a single scan starting at $\mathsf{EQ}[|S|]$.

The events are processed in order of the priority and for events with the same priority they are processed in the order as above. Events of the same type and with the same priority are processed in arbitrary order. In the following we only look at one edge at the time when events of type 1, 2, and 3 are taken care of. Due to space limitations many algorithmic details are left out in the following. See [5] for a detailed description of the algorithm.

1. Excess change. The excess changes for all chunks at the same time, namely when $|\alpha| = i \cdot p(\alpha)$ for $i = 2, 3, 4, \ldots$. For each chunk in $\mathsf{CHUNKLIST}(\alpha)$ we will remove the chunk from $\mathsf{D}(\alpha)$, recompute the excess and nominal contribution based on the number of occurrences in the chunk, update $\mathsf{NCS}(\alpha)$, reinsert the chunk with the new excess and finally update $\mathsf{ZS}(\alpha)$. This is done as follows:

First decide which chain each chunk belongs to by searching the tree. Remove each chunk from its chain by splitting the tree for the chain. Recompute the excess for each chunk and reconstruct the tree. In the new tree the chain structure may have changed. Chunks for which the excess increases to two will be separate chains, while chunks where the excess become less than two may join two or three chains into one chain. $\mathsf{NCS}(\alpha)$ and $\mathsf{ZS}(\alpha)$ are always kept updated during the processing of the event.

If $|\alpha| = 2p(\alpha)$ then insert an event of type 2 with priority $2p(\alpha)$ into EQ, with a pointer to any leaf in $\mathsf{D}(\alpha)$. If $|\alpha| = ip(\alpha) > 2p(\alpha)$, then insert an event of type 1 with priority $(i-1)p(\alpha)$ into EQ, with a pointer to any leaf in $\mathsf{D}(\alpha)$.

2. Chunks become necklaces. When $|\alpha|$ decreases to $2p$ all chunks become necklaces at the same time. At this point all chunks and necklaces that overlap shall be joined into one necklace. Note that all chunks have excess 0 or 1 when $|\alpha| = 2p$ and since we first recompute the excess all overlapping chunks and necklaces are in the same chain. Hence, what we have to do is to join all chunks and necklaces from left to right, in each chain.

This is done by first deciding for each chunk which chain it belongs to. Next, for each chain containing at least one chunk, join all chunks and necklaces from left to right. Update $\mathsf{NCS}(\alpha)$ and $\mathsf{ZS}(\alpha)$.

3. Necklace and chain break-up. When two consecutive occurrences of α with indices e_1 and e_2 terminate to overlap this may cause a necklace or a chain to break up into two necklaces or chains.

If e_1 and e_2 belong to the same chain then the chain breaks up in two chains. If e_1 and e_2 belongs to the same necklace then split both the necklace and the chain between e_1 and e_2. If e_1 and e_2 belong to different necklaces or chunks in the chain then split the chain between the two subtrees including e_1 and e_2 respectively. Update $\mathsf{NCS}(\alpha)$ and $\mathsf{ZS}(\alpha)$.

4. Merging at internal nodes. Let α be a substring such that the locus of α is a node v in the suffix tree. Then the leaf-list, $\mathsf{LL}(v)$ for v is the union of the leaf-lists for the subtrees below v, hence at the nodes in the suffix tree the data structures for the subtrees should be merged into one. We assume that the edges below v are processed for α as described above.

Let $T_1, \ldots, T_t$ be the subtrees below v in the suffix tree. We never merge more than two data structures at the time. If there are more than two subtrees the merging is done in the following order: $T = \mathsf{Merge}(T, T_i)$, for $i = 2, \ldots, t$, where $T = T_1$ to start with. This can also be viewed as if the suffix tree is made binary by replacing all nodes of degree larger than 2 by a binary tree with edges without labels. From now on we will describe how to merge the data structures for two subtrees.

The merging will be done by inserting all indices from the smaller of the two leaf-lists into the data structure for the larger one. Let T denote the 3-level search tree to insert new indices in and denote by $e_1, \ldots, e_m$ the indices to insert, where $e_i < e_{i+1}$. The insertion is done by first splitting the tree T at all positions e_i for $i = 1, \ldots, m$. The tree is then reconstructed from left to right at the same time as the new indices are inserted in increasing order. Assume that the tree is reconstructed for all indices, in both trees, smaller than e_i. The next step is to insert e_i and all indices between e_i and e_{i+1}. This is done as follows:

Check if the occurrence with index e_i overlaps any occurrences to the left, i.e. an occurrence in the tree reconstructed so far. Insert e_i into the tree. If e_i overlaps with an occurrence already in the tree then check in what way this affects the chain, chunk, and necklace structure and do the appropriate updates. Do the corresponding check and updates when the tree to the right of e_i (the tree for indices between e_i and e_{i+1}) is incorporated, i.e. check if e_i will cause any further changes in the chain, chunk, and necklace structure due to overlaps to the right. Update $\mathsf{NCS}(\alpha)$ and $\mathsf{ZS}(\alpha)$.

Every time, during the above described procedure, when two overlapping occurrences with indices e_i and e_j, $e_i < e_j$, from different subtrees are encountered the event (e_i, e_j) with priority $e_j - e_i$ is inserted into the event queue EQ and the previous event, if any, with a pointer to e_i is removed from EQ. Update $p(\alpha)$ to $e_j - e_i$ if this is smaller than the current $p(\alpha)$ value. If $|\alpha| > 2p(\alpha)$ then insert an event of type 1 with priority $\lfloor |\alpha|/p(\alpha) \rfloor p(\alpha)$ into EQ, with a pointer to any leaf in $\mathsf{D}(\alpha)$.

6 Analysis

Theorem 3. *The minimal augmented suffix tree,* $\mathsf{MAST}(S)$*, for a string* S *of length* n *can be constructed in time* $\mathcal{O}(n \log n)$ *and space* $\mathcal{O}(n)$.

In the full version of the paper [5] we show that the running time of the algorithm in Section 5 is $\mathcal{O}(n \log n)$. Here we only state the main steps of the proof. The proof uses an amortization argument, allowing each edge to be processed in amortized constant time, and each binary merge at a node (in the binary version) of $\mathsf{ST}(S)$ of two leaf-lists of sizes n_1 and n_2, with $n_1 \geq n_2$, in amortized time $\mathcal{O}(n_2 \log \frac{n_1+n_2}{n_2})$. From Lemma 1 it then follows that the total time for processing the internal nodes and edges of $\mathsf{ST}(S)$ is $\mathcal{O}(n \log n)$.

Using Theorem 1 and 2 we can prove that: Processing events of types 1 and 2 take time $\mathcal{O}(m \log \frac{|\mathsf{LL}(v)|}{m})$, where $m = |\mathsf{CHUNKLIST}(\alpha)|$. Processing an event of type 3 takes time $\mathcal{O}(\log |c|)$, where c is the chain being split. An event of type 4 has processing time $\mathcal{O}(n_1 \log \frac{n_1+n_2}{n_1})$.

Let v be a node in the suffix tree and let α be a string with locus v or locus on the edge immediately above v. For the data structure $\mathsf{D}(\alpha)$ we define a potential $\Phi(\mathsf{D}(\alpha))$. Let $\mathcal{C}$ be the set of chains stored in $\mathsf{D}(\alpha)$, and for a chain c let $|c|$ denote the number of occurrences of α in c. We define the potential of $\mathsf{D}(\alpha)$ by $\Phi(\mathsf{D}(\alpha)) = \Phi_1(\alpha) + \Phi_2(\alpha) + \sum_{c \in \mathcal{C}} \Phi_3(c)$, where the rôle of Φ_1, Φ_2, and Φ_3 is to account for the potential required to be able to process events of type 1, 2, and 3 respectively. For a chunk, with leftmost occurrence of α at position i, consider the substring $S[i\mathinner{..}j]$ with maximal j and $S[i\mathinner{..}j]$ having period p, where $p = p(\alpha)$ is the period of α. We denote the chunk *green* if and only if $|\alpha| \bmod p \leq j - i + 1 \bmod p$. Otherwise the chunk is *red*. Let k denote the number of chunks in $\mathsf{D}(\alpha)$ and let g denote the number of green chunks in $\mathsf{D}(\alpha)$.

We define $\Phi_1(\alpha) = 7g \log \frac{|v| \cdot e}{g}$, $\Phi_2(\alpha) = k \log \frac{|v| \cdot e}{k}$, and $\Phi_3(c) = 2|c| - \log |c| - 2$, with the exceptions that $\Phi_1(\alpha) = 0$ if $g = 0$, and $\Phi_2(\alpha) = 0$ if $k = 0$.

We can prove that processing events of type 1, 2, and 3 release sufficient potential to pay for the processing, while processing an event of type 4 increases the potential by $\mathcal{O}(n_1 \log \frac{n_1+n_2}{n_1})$. By Lemma 1 the total amortized time for handling all events is $\mathcal{O}(n \log n)$.

References

1. A. Apostolico and A. Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theoretical Computer Science*, 119:247–265, 1993.
2. A. Apostolico and F. P. Preparata. Optimal off-line detection of repetitions in a string. *Theoretical Computer Science*, 22:297–315, 1983.
3. A. Apostolico and F. P. Preparata. Data structures and algorithms for the string statistics problem. *Algorithmica*, 15:481–494, 1996.
4. G. S. Brodal, R. Lyngsø, C. N. S. Pedersen, and J. Stoye. Finding maximal pairs with bounded gap. *Journal of Discrete Algorithms, Special Issue of Matching Patterns*, 1(1):77–104, 2000.
5. G. S. Brodal, R. B. Lyngsø, A. Östlin, and C. N. S. Pedersen. Solving the string statistics problem in time $O(n \log n)$. Technical Report RS-02-13, BRICS, Department of Computer Science, University of Aarhus, 2002.
6. G. S. Brodal and C. N. S. Pedersen. Finding maximal quasiperiodicities in strings. In *Proc. 11th Combinatorial Pattern Matching*, volume 1848 of *Lecture Notes in Computer Science*, pages 397–411. Springer Verlag, Berlin, 2000.
7. M. R. Brown and R. E. Tarjan. A fast merging algorithm. *Journal of the ACM*, 26(2):211–226, 1979.
8. M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 137–143, 1997.
9. A. S. Fraenkel and J. Simpson. How many squares can a string contain? *Journal of Combinatorial Theory, Series A*, 82(1):112–120, 1998.
10. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
11. K. Hoffmann, K. Mehlhorn, P. Rosenstiehl, and R. E. Tarjan. Sorting Jordan sequences in linear time using level-linked search trees. *Information and Control*, 86(1-3):170–184, 1986.
12. S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.
13. F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal of Computing*, 1(1):31–39, 1972.
14. D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6:323–350, 1977.
15. E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
16. K. Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer Verlag, Berlin, 1984.
17. J. Stoye and D. Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. *Theoretical Computer Science*, 270:843–856, 2002.
18. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.
19. P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Symposium on Switching and Automata Theory*, pages 1–11, 1973.