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

Gerth Stølting Brodal[†,‡] Rune B. Lyngsø[§] Anna Östlin[¶]
Christian N. S. Pedersen[†,‖]

October 10, 2002

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.

Keywords: Strings, suffix trees, string statistics, periods, search trees

*A short version of this paper has been published as [5]

[†]BRICS (Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation), Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Århus C, Denmark. E-mail: {gerth,cstorm}@brics.dk. 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).

[§]Department of Statistics, Oxford University, Oxford OX1 3TG, UK. E-mail: lyngsoe@stats.ox.ac.uk.

[¶]IT University of Copenhagen, Glentevej 67, DK-2400 Copenhagen NV. E-mail: annao@it-c.dk. Work done while at BRICS.

[‖]Bioinformatics Research Center (BiRC), www.birc.dk, funded by Aarhus University Research Foundation.

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.

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 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, 18], 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.

The rest of this section contains basic definitions and lemmas that we will use in the latter sections. In Section 2 we give a precise definition of the string statistics problem and $\mathsf{MAST}(S)$. In Section 3 we give all the string properties and definitions enabling us to construct $\mathsf{MAST}(S)$ in time $\mathcal{O}(n \log n)$, and a self-contained proof of the theorem of Fraenkel and Simpson that any string

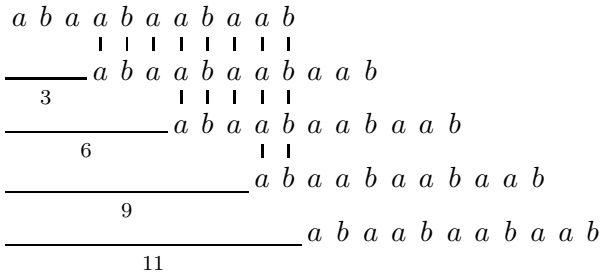


Figure 1: The string *abaabaabaab* has periods 3, 6, 9, and 11, i.e. the period of the string is 3

has at most a linear number of distinct squares as substrings. In Section 4 we describe the variant of level-linked (2,4)-trees used by our algorithm. In Section 5 we present our algorithm, and in Section 6 we prove that the running time of the algorithm is $\mathcal{O}(n \log n)$.

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 $p \leq |S|/2$. Figure 1 shows the periods of a string of length 11. 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 $\underline{b\,a\,b}\,a\,a\,a\,\underline{b\,a\,b}\,a\,a\,b$ 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 $\underline{b\,a\,b}\,\underline{a}\,\underline{b\,a\,b}\,\underline{a}\,\underline{b\,a\,b}$ 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.*

Figure 2 shows the suffix tree of a string of length 13. 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, 16, 19, 20]. 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 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 $\mathcal{O}(|\text{small}(v)| \cdot \log(|v|/|\text{small}(v)|))$. Lemma 1 then states that the total time becomes $\mathcal{O}(n \log n)$ [17, Exercise 35].

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

$$\sum_{v \in T} c_v \leq n \log n \ .$$

Proof. The proof is by induction in the size of T. If $|T| = 1$, then the lemma holds vacuously. Now assume inductively that the upper bound holds for all trees with at most $n-1$ leaves. Consider a tree with n leaves where the number of leaves in the subtrees rooted at the two children of the root are k and $n-k$ where $0 < k \leq n/2$. According to the induction hypothesis the sum over all nodes in the two subtrees, is bounded by respectively $k \cdot \log k$ and $(n-k) \cdot \log(n-k)$. The entire sum is thus bounded by $k \log(n/k) + k \log k + (n-k) \log(n-k) = k \log n + (n-k) \log(n-k) < n \log n$, which proves the lemma. □

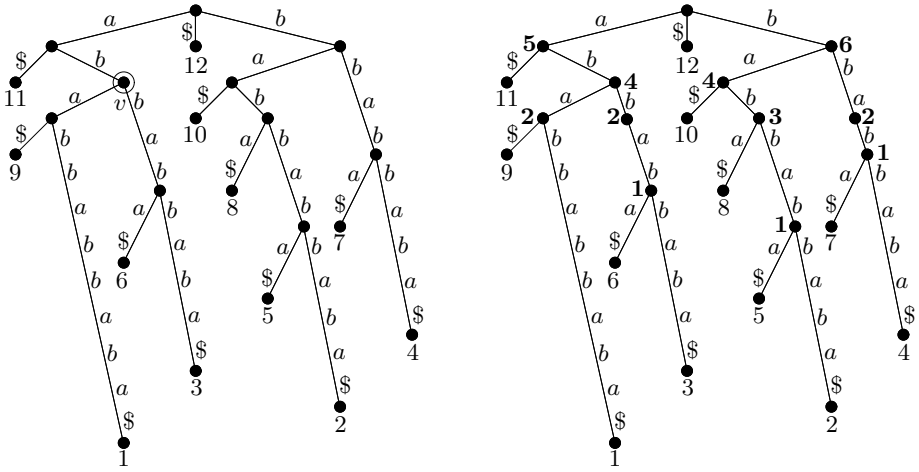


Figure 2: To the left is the suffix tree $\mathsf{ST}(S)$ of the string $S = a\,b\,a\,b\,b\,a\,b\,b\,a\,b\,a$. The node v has path-label $\mathsf{L}(v) = a\,b$ and leaf-list $\mathsf{LL}(v) = \{1, 3, 6, 9\}$. To the right is the minimal augmented suffix tree $\mathsf{MAST}(S)$ for the string $S = a\,b\,a\,b\,b\,a\,b\,b\,a\,b\,a$. Internal nodes are labelled with the c-values.

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].

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 2 shows the suffix tree and the minimal augmented suffix tree for the string $a\,b\,a\,b\,b\,a\,b\,b\,a\,b\,a$.

The space needed to store $\mathsf{MAST}(S)$ is $\mathcal{O}(n)$, since by Lemma 6 the minimal augmented suffix tree has at most $3n$ internal nodes.

3 String Properties

Lyndon and Schutzenberger [15] proved the following periodicity lemma for periodic strings.

Lemma 2 *If a string S has two periods $p, q \leq |S|/2$, then $\gcd(p, q)$ is also a period of S.*

If S is periodic, then by Lemma 2 the period p of S divides all periods of S less than or equal to $|S|/2$. Any prefix S' of S with length at least p also has period p. If S' has length at least $2p$, then the period of S' by Lemma 2 divides p, implying that the period of S' also is a period of S.

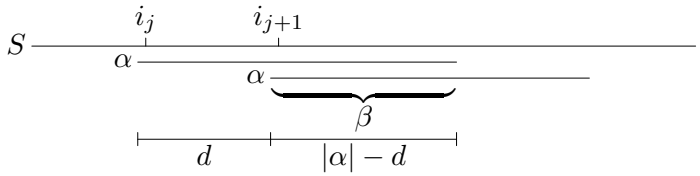
Corollary 1 *If S has period $p \leq |S|/2$, then p is also the period of the prefixes $S[1\,..\,k]$ for $2p \leq k \leq |S|$.*

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

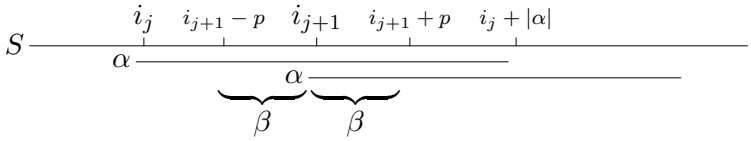
Lemma 3 *Let S be a string and α a substring of S. If the occurrences of α in S are at positions $i_1 < \cdots < i_k$, then for all $1 \leq j < k$ either $i_{j+1} - i_j = p$ or $i_{j+1} - i_j > \max\{|\alpha| - p, p\}$, where p denotes the period of α.*

Proof. Consider two consecutive and overlapping occurrences of α at positions i_j and i_{j+1} in S, i.e. there are no occurrences of α at position k for $i_j < k < i_{j+1}$. Let $d = i_{j+1} - i_j \geq 1$. We will show that neither of the two cases $d < p$ or $p < d \leq |\alpha| - p$ are possible. The two cases are illustrated in Figure 3.

If $d < p$, then $d < |\alpha|$ since $p \leq |\alpha|$. By definition α occurs at positions i_j and $i_j + d$, implying $\beta = \alpha[1\,..\,|\alpha| - d]$ is both a prefix and suffix of α. See Figure 3(a). By definition, d is then a period of α, contradicting that p is the shortest period of α.



(a) The case $d < p$



(b) The case $p < d \leq |\alpha| - p$

Figure 3: The two cases considered in the proof of Lemma 3

If $p < d \leq |\alpha| - p$, then we have the inequalities $i_j < i_{j+1} - p < i_{j+1} < i_{j+1} + p \leq i_j + |\alpha|$. If α has period p, then α is a prefix of the infinite string $\beta\beta\beta\ldots$, where $\beta = \alpha[1\,..\,p]$. It follows that $\beta = S[i_{j+1}\,..\,i_{j+1} + p - 1] = S[i_{j+1} - p\,..\,i_{j+1} - 1]$, implying that α occurs at position $i_{j+1} - p$, which contradicts the assumption that there is no occurrence of α between positions i_j and i_{j+1}. □

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 2 *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 4 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.

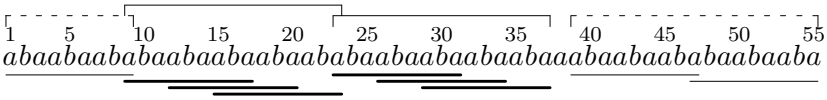


Figure 4: The grouping of occurrences in a string into chunks and necklaces. Occurrences are shown below the string. Thick lines are occurrences in chunks. The grouping into chunks and necklaces is shown above the string. Necklaces are shown using dashed lines. Note that a necklace can consist of a single occurrence

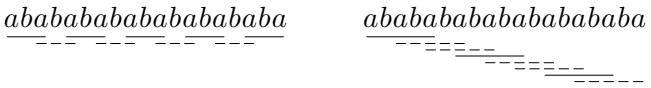


Figure 5: Examples of the contribution to the c-values by an isolated necklace (left; $\alpha = aba$ and the contribution is $5 = \lceil 9/2 \rceil$) and an isolated chunk (right; $\alpha = ababa$, $p = 2$, and the contribution is $3 = \lceil 8/\lceil 5/2 \rceil \rceil$)

In Figure 4 the chunk covering positions 9–23 overlaps with the necklace covering positions 1–9, and the chunk covering positions 23–37.

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 5 gives an example of the contribution to the c-values by an isolated necklace and chunk.

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.*

Proof. Since only two consecutive occurrences in a necklace overlap, exactly every second occurrence in the necklace can contribute to the c-values (see Figure 5 (left)).

In a chunk of k occurrences all occurrences have distance p, implying that only every $\lceil |\alpha|/p \rceil^{\text{th}}$ occurrence contributes to the c-value of α and the stated contribution follows (see Figure 5 (right)). □

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. The actual contribution to the c-value of α as a result of applying the greedy algorithm can at most be one less for each necklace and chunk as argued in the proof of Lemma 5 below.

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 $\alpha[1\,..\,p]$ 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].

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*

$$\sum_{c \in \mathcal{C}} \lfloor \#_0(c)/2 \rfloor \ .$$

Proof. We consider the maximum number of non-overlapping occurrences generated by the greedy algorithm described in Section 2.

Let b be a chunk or a necklace of k occurrences, and let $z = \lceil |\alpha|/p \rceil$ if b is chunk, and $z = 2$ if b is a necklace. From Lemma 3 only the first occurrence in b can overlap with the occurrence immediately preceding b, and the overlap is at most $p-1$ positions. If there is no immediate preceding occurrence that overlaps with b, or the immediate preceding occurrence is not reported by the greedy algorithm, then the contribution to the c-value of b equals the nominal contribution. If the immediate preceding occurrence of b was reported and overlaps with the first occurrence in b, then the $2 + i \cdot z^{\text{th}}$ occurrences in b are reported for $i = 0, \ldots, \lceil (k-1)/z \rceil - 1$, i.e. the number of non-overlapping occurrences reported in b equals the nominal contribution if and only if $\lceil k/z \rceil = \lceil (k-1)/z \rceil$, i.e. b has nonzero excess. If the excess is zero, then the contribution of the chunk is one less than the nominal contribution. It remains to count how often this last case happens.

Consider a chain consisting of a single chunk with excess at least two. Then the contribution of the chain equals the nominal contribution and $\#_0(c) = 0$. None of the two possible ways to report occurrences in the chunk by the greedy algorithm includes the last occurrence of the chunk. This implies that the reporting within a chunk or necklace b of excess zero is only influenced by the necklaces and chunks to the left of b contained within the same chain as b.

Consider a chain c where all necklaces and chunks have excess zero or one. For all chunks and necklaces in the chain with excess one, the last occurrence in the chunk or necklace is reported if and only if the last occurrence in the immediate preceding chunk or necklace in the chain is reported. It follows that the last occurrence is reported for every second chunk or necklace in the chain with excess zero, including the first with excess zero. We conclude that the number of chunks and necklaces with excess zero in the chain c where the contribution to the c-value of α is one less than the nominal contribution is $\lfloor \#_0(c)/2 \rfloor$. □

To bound the size of a minimal augmented suffix tree we need the following theorem of Fraenkel and Simpson, who proved that a string can at most contain a linear number of distinct squares [9]. The proof given below is a slight simplification of [9, Theorem 1].

Theorem 1 (Fraenkel and Simpson) *The number of distinct squares occurring in a nonempty string S is less than $2|S|$.*

Proof. We prove that any position i in S can at most be the rightmost occurrence of two distinct squares in S. Assume for the sake of contradiction that i is the rightmost occurrence of three squares $\alpha\alpha$, $\beta\beta$ and $\gamma\gamma$ with $a = |\alpha|$, $b = |\beta|$, $c = |\gamma|$ and $0 < a < b < c$. Without loss of generality we assume $s = \gamma\gamma$ and $i = 1$. Figure 6 shows the two cases to be considered.

If $a \leq c/2$, then $\alpha\alpha$ is a prefix of γ. Therefore $\alpha\alpha$ also occurs at positions $c + 1$, which contradicts the assumption that position one is the rightmost occurrence of $\alpha\alpha$.

If $a > c/2$, then we will show that $\alpha\alpha$ occurs at position $p+1$, where p is the period of α. By Lemma 3 the occurrences of α at positions $a+1$, $b+1$, and $c+1$ will have distance at least p (α is a prefix of β and γ, which occur respectively at positions $b + 1$ and $c + 1$), i.e. $a + p \leq b \leq c - p$. Since α has period p and occurs at positions one and $a + 1$, it follows that α occurs at positions $p + 1$ and $a+p+1$ if $S[a-p+1\,..\,a+p]$ and $S[2a-p+1\,..\,2a+p]$ are substrings of α. We have $S[a-p+1\,..\,a+p] = \beta[a-p+1\,..\,a+p] = S[a+b-p+1\,..\,a+b+p] = \alpha[a+b-c-p+1\,..\,a+b-c+p]$ since $a+b-c-p+1 \geq a+a+p-c-p+1 > 1$ and $a+b-c+p \leq a$, and $S[2a-p+1\,..\,2a+p] = \beta[2a-b-p+1\,..\,2a-b+p] = \alpha[2a-b-p+1\,..\,2a-b+p]$ since $2a-b-p+1 > c-b-p+1 \geq p-p+1 = 1$ and $2a-b+p \leq 2a-(a+p)+p = a$. We conclude that $\alpha\alpha$ occurs at position $p+1$,

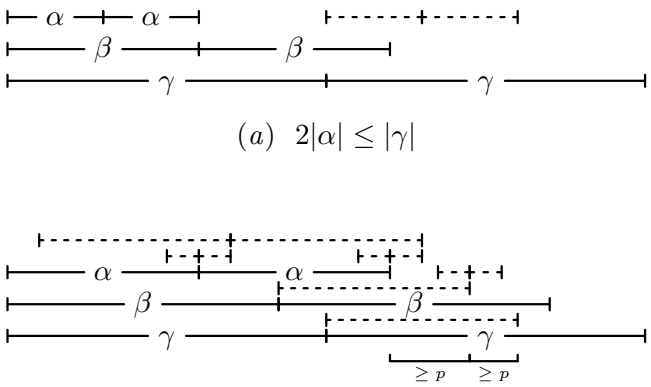


Figure 6: The two cases in the proof of Theorem 1. Long dashed lines are occurrences of α and short dashed lines are occurrences of $\alpha[1\,..\,p]$

which contradicts the assumption that the rightmost occurrence of $\alpha\alpha$ is at position one. Since no square starts at position $|S|$ the theorem holds. □

Lemma 6 *The minimal augmented suffix tree for a string S has at most $3|S|$ internal nodes.*

Proof. Since the suffix tree for a string S has $|S|$ internal nodes we only need to show that there are at most $2|S|$ extra nodes in the minimal augmented suffix tree. If the number of extra nodes are limited by the number of squares in the string the thesis follows from Theorem 1.

Extra nodes are only located on edges when there is a change in the c-value along that edge. The c-value changes due to two reasons. First, when two substrings become equal as they get shorter and second, when two occurrences of a substring no longer overlap as they get shorter. The first reason only gives rise to changed c-values in nodes already in the suffix tree. If α occurs at position i and $i + |\alpha| - 1$ both occurrences can not be counted in the c-value of α. However, the occurrences of $\alpha' = \alpha[1..|\alpha| - 1]$, at positions i and $i + |\alpha| - 1$, do not overlap and the c-value may change. This only happens when there is a square $\alpha'\alpha'$ in S. □

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 [17, Section III.5]. The operations we consider supported are:

NewTree(e, w): Creates a new tree T containing the element e with associated weight w.

Search(p, e): Searches 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.

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.

Delete(p): Deletes the leaf and element that p is a pointer to in a tree T.

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.

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.

Weight(T): Returns the sum of the weights of the elements in the tree T.

Hoffman et al. [11, Section 3] considered the case where elements are unweighted, and showed how level-linked (2,4)-trees support all the above operations, except Weight, within the time bounds stated in Theorem 2 below.

Theorem 2 (Hoffmann et al.) *Level-linked (2,4)-trees support the operations* 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, in the following denoted *extreme nodes*. 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 3 *Weighted level-linked (2,4)-trees support the operations* 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* $\mathcal{O}(\log \min\{|T_1|, |T_2|\})$*.*

Proof. The operations NewTree and Weight take amortized constant time, since the total weight of a tree is explicitly stored, and Search takes time $\mathcal{O}(\log d)$ since it is unaffected by the presence of weights. The presence of weights increases the time for Insert and Delete to $\mathcal{O}(\log |T|)$, since in the worst-case all weights stored on the path from the root to the new/deleted leaf must be recomputed for every insertion and deletion.

For the operations Join and Split we need to argue that the stored weights can be updated within the claimed time bounds. The Join operation proceeds by first linking the root of the tree of minimal height $h = \mathcal{O}(\log \min\{|T_1|, |T_2|\})$ as a child of an extreme node with height $h + 1$, followed by a sequence of node splittings. The linking causes $2h - 1$ extreme nodes not to be extreme nodes any longer, forcing the weights to be computed for these nodes. This can be done in time $\mathcal{O}(h)$. Since for each node splitting the weights can be updated in constant time and the total weight of T is the sum of the weights of T_1 and T_2, the amortized time bound for Join follows. A Split consists of a sequence of node splittings, unlinking the subtree rooted a node of height $h = \mathcal{O}(\log \min\{|T_1|, |T_2|\})$, and a sequence of node fusions. The weights at the involved nodes can similarly to Join be updated during the sequence of node splittings and fusions. The unlinking only creates new extreme nodes. Finally the weight of the resulting tree of minimal height can be computed by traversing the extreme nodes of this tree in time $\mathcal{O}(h)$. The weight of the larger tree can be computed as the difference between the weight of T and the weight of the tree of minimal height. □

5 The Algorithm

In this section we describe the algorithm for constructing the minimal augmented suffix tree for a string S of length n. The analysis is presented in Section 6 and shows that the algorithm runs in time $\mathcal{O}(n \log n)$.

5.1 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

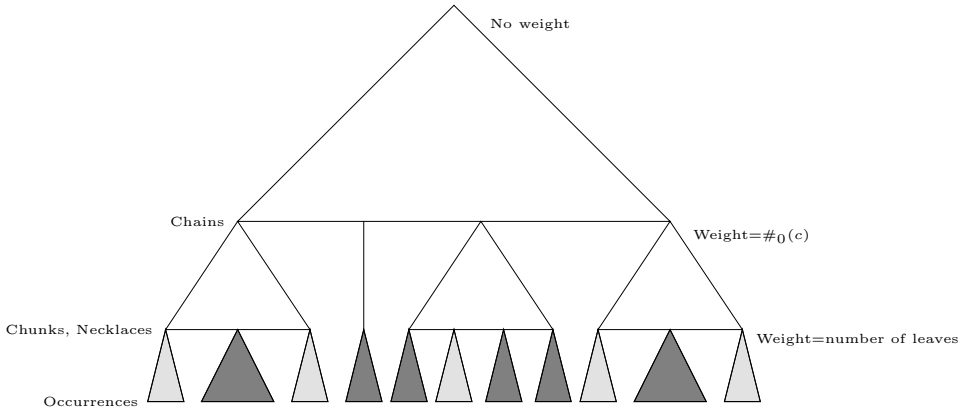


Figure 7: The data structure is a 3-level search tree.

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 compute efficiently 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.

5.2 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. See Figure 7.

Top level: Unweighted (2,4)-tree (cf. Theorem 2) with the chains as leaves. The key of a chain is the leftmost index in the chain.

Middle level: One weighted (2,4)-tree (cf. Theorem 3) for each chain, with the chunks and necklaces as leaves. The leftmost indices in the chunks and necklaces 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 (cf. Theorem 2) for each chunk and necklace, with the occurrences in the chunk or necklace as the leaves.

The weight of a leaf is 1. 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 as defined in Section 3 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 chains containing at least one chunk, denoted $\mathsf{CHAINLIST}(\alpha)$. The list will keep pointers to the roots of the trees for the chains, and pointers in the opposite direction are kept in the tree. $\mathsf{CHAINLIST}(\alpha)$ will be useful when all chunks are processed at the same time and we do not want to spend time on searching for chunks in chains with no chunks. 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 2, $p(\alpha)$ is the period of α if there is at least one chunk. For convenience, 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 2, and for the middle and bottom level trees in $\mathsf{D}(\alpha)$ we will use weighted level-linked (2,4)-trees, according to Theorem 3. 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.

5.3 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 $2p(\alpha)$ 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. Necklaces and chains 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, where $e_1 < e_2$, terminate to overlap, store a pointer to the leaf e_1 in the suffix tree. For the leaf e_1 in the suffix tree a pointer to the event in EQ is also 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 decreasing 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.

5.3.1 Excess change

The excess changes for all chunks at the same time, namely when $|\alpha| = i \cdot p(\alpha)$ for $i = 2, 3, 4, \ldots$. To update the data structure when the excess changes, all chunks are first removed from $\mathsf{D}(\alpha)$. Then the new excess and nominal contribution are computed and finally the tree is reconstructed as the removed chunks are reinserted into $\mathsf{D}(\alpha)$. This is all done for each chain in $\mathsf{CHAINLIST}(\alpha)$ seperately. In details the following is done.

1. For each chain in $\mathsf{CHAINLIST}(\alpha)$ find the sorted list of chunks in the chain. Note that at least every other element in the chain are chunks, since two necklaces do not overlap.

2. For each chunk remove it from its chain by splitting the tree for the chain. At most two Split-operations per chunk are needed to temporary make each chunk a chain by itself. The chain which the chunk is part of is replaced by the at most three new (overlapping) chains in the top-level tree, unless the chunk already was a chain by itself. Keep $\mathsf{CHAINLIST}(\alpha)$ updated by deleting the split chain from the list and inserting the resulting chains, if they contain at least one chunk. Keep $\mathsf{ZS}(\alpha)$ updated by first subtracting the old contribution of the chain to $\mathsf{ZS}(\alpha)$ and then adding the contributions of the resulting chains.

3. Recompute the excess and nominal contribution for all chunks and update $\mathsf{NCS}(\alpha)$ accordingly by adding the change in nominal contribution to $\mathsf{NCS}(\alpha)$ for each chunk. Set the weight of chunks with excess 0 to 1 and set the weight of chunks with excess at least 1 to 0. In the new tree the chain structure may have changed. Chunks for which the excess increases to two or more will become separate chains, while chunks where the excess becomes less than two may join two or three chains into a single chain.

4. Now we want to reconstruct the tree by reinserting the chunks into the tree. This is done seperately for each chain in the original $\mathsf{CHAINLIST}(\alpha)$. Let $c_1, \ldots, c_m$ denote the chunks from left to right in a chain and let T_i denote the tree for the chain containing chunk c_i and do the following.

   If c_i has excess 2 or more then nothing is done since c_i is a chain by itself. If the excess is 0 or 1 we check if the chunk overlaps the chains to the left and right, denoted T_ℓ and T_r, and in that case if the chains shall be joined.

   (a) If c_i to the left overlaps with a necklace or a chunk with excess 0 or 1, then remove T_ℓ from the top-level tree and join T_i and T_ℓ. Denote the resulting tree T_i. Delete from $\mathsf{CHAINLIST}(\alpha)$ the chain containing c_i and also the chain T_l if it is in the list. Insert the new chain into $\mathsf{CHAINLIST}(\alpha)$. Update $\mathsf{ZS}(\alpha)$ accordingly.

   (b) If T_r to the right overlaps with a necklace then join T_i with T_r after removing T_r from the top-level tree. Denote the resulting tree T_i. Update $\mathsf{ZS}(\alpha)$ accordingly.

   If $T_r = T_{i+1}$, i.e. it is also a chunk, then the overlap between T_i and T_{i+1} will be checked as described above when continuing with T_{i+1}.

Continue until all chunks are checked and repeat the same procedure for each chains in the original $\mathsf{CHAINLIST}(\alpha)$.

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

5.3.2 2. Chunks become necklaces.

When $|\alpha|$ decreases to $2p(\alpha)$ all chunks become necklaces at the same time. At this point all chunks and necklaces that overlap must be joined into one necklace. Note that all chunks have excess 0 or 1 when $|\alpha| = 2p(\alpha)$ 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. The following is done.

1. For each chain in $\mathsf{CHAINLIST}(\alpha)$ join all chunks and necklaces from left to right. Note that there are never two consecutive necklaces in the same chain, i.e. there are at most two Join-operations per chunk. Update $\mathsf{NCS}(\alpha)$ and $\mathsf{ZS}(\alpha)$ by adding the difference in nominal contribution to $\mathsf{NCS}(\alpha)$ and by subtracting $\#_0(c)$ from $\mathsf{ZS}(\alpha)$ for each changed chain and add it for the new chains.

2. Set $\mathsf{CHAINLIST}(\alpha)$ to be empty, since there are no chunks left.

5.3.3 Necklaces and chains 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 belong to the same necklace then both the necklace and the chain split between e_1 and e_2. The following is done to update the data structure.

1. Using the pointers stored at the leaves of $\mathsf{ST}(S)$, check if the indices e_1 and e_2 are already in different chains. If they are then nothing is done.

2. If e_1 and e_2 are in the same chain, represented in the tree by T_c, then c shall be split.

   (a) If both indices belong to the same necklace then the necklace shall be split. Remove the tree for the necklace from T_c by performing two Split-operations. The result is three trees, T_{c_1}, T_n, and T_{c_2}. Split the tree for the necklace into two parts by performing

$\mathsf{Split}(T_n, e_2) = T_{n_1}, T_{n_2}$. Insert T_{n_1} in T_{c_1} and T_{n_2} in T_{c_2} using two Join-operation. Finally, insert the new chain into the top-level tree and update $\mathsf{NCS}(\alpha)$ and $\mathsf{ZS}(\alpha)$.

(b) If the two indices belong to two different subtrees at the bottom-level then only the chain has to be split. Insert the new chain into the top-level tree and update $\mathsf{ZS}(\alpha)$.

Update $\mathsf{CHAINLIST}(\alpha)$ if necessary, by first removing c from the list (if it is in it). Then for the two new chains, check if they contain at least one chunk and in that case insert the chains into $\mathsf{CHAINLIST}(\alpha)$.

5.3.4 Merging at internal nodes

Let α be a substring such that the locus of α is a node v in the suffix tree. The leaf-list for v, $\mathsf{LL}(v)$, 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 have been processed for α as described above for events 1, 2, and 3.

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 with empty labels. We will now 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 increasing $i = 1, \ldots, m$. The tree is then reconstructed, from left to right, at the same time as the new indices are inserted. More exactly the following is done.

1. For all indices e_i, $i = 1, \ldots, m$ split T (at all levels necessary) by performing $\mathsf{Split}(e_i)$. Denote the resulting 3-level trees $T_0, \ldots, T_{m+1}$ where e_i is larger than all indices in T_j for $j < i$, and smaller than all other indices. Note that T_i is an empty tree if there are no indices in T between e_i and e_{i+1}.

   When a middle-level tree, for a chain c, is split, update $\mathsf{ZS}(\alpha)$ by subtracting $\#_0(c)$ and adding the new $\#_0$-values for the two new trees. Update $\mathsf{CHAINLIST}(\alpha)$ (from the tree T) if necessary, by first removing c from the list (if it is in it). Then for the two new chains, check if they contain at least one chunk and in that case insert the chains into

$\mathsf{CHAINLIST}(\alpha)$. When a bottom-level tree is split, update $\mathsf{NCS}(\alpha)$ by subtracting the nominal contribution of the split chunk or necklace and adding the two new nominal contributions.

2. The tree is reconstructed from left to right by inserting the new indices in increasing order. Let T' be the tree reconstructed for all indices smaller than e_i in the two trees to be merged, i.e. T' is the union of $e_0, \ldots, e_i$ and $T_0, \ldots, T_i$. Initially T' equals T_0. To proceed, we want to insert e_i and all indices between e_i and e_{i+1} into T'. (If $i = m$ we insert all remaining indices.) This is done as follows.

   Check if the occurrence with index e_i overlaps the occurrence for the rightmost index in the tree T' reconstructed so far and in that case, check if this index is part of a necklace or a chunk.

   (a) If the occurrence with index e_i does not overlap any occurrence to the left then create a new chain with one necklace with e_i as the only index. Insert the chain into T'.

   (b) If the occurrence with index e_i overlaps an occurrence with index e_ℓ to the left and the overlap is less than $|\alpha|/2$ then if e_ℓ is part of a necklace then insert e_i into this necklace using one Join-operation. If e_ℓ is part of a chunk, c_ℓ, then create a new chain with one necklace with e_i as the only index. If the excess of c_ℓ is 2 or more then insert the chain in T', otherwise join the two chains.

   (c) If the occurrence with index e_i overlaps an occurrence with index e_ℓ to the left and the overlap is more than $|\alpha|/2$ then if e_ℓ is part of a necklace remove e_ℓ from the necklace and create a new chunk with e_ℓ and e_i as the two indices. Create a new chain for the chunk with two indices e_ℓ and e_i. Since the excess of the chunk with 2 indices is 1, join the two chains. If e_ℓ is part of a chunk then insert e_i into the chunk. If the excess increases from 1 to 2 then let the chunk be a chain by itself. If the excess changes to 0 after being 2 or more then join the chunk with the chain to the left if possible.

   The tree T' is at this point the tree reconstructed for all indices up to index e_i. The next step is to include also the tree T_i. Unless T_i is empty the following is done.

   Check if the occurrence with index e_i overlaps the leftmost occurrence in T_i, with index e_r . If it does not we just join the trees T' and T_i. If it does overlap the following is done.

   (a) The overlap between e_i and e_r is less than $|\alpha|/2$. If both e_i and e_r are part of necklaces then join the necklaces and the chains they are part of. If at least one of e_i and e_r is part of a chunk, then join

the chains they are part of unless the excess of the chunk is 2 or more.

(b) The overlap between e_i and e_r is more than $|\alpha|/2$. If both e_i and e_r are part of necklaces then remove e_i and e_r from their trees and create a new chunk with the two indices. Since the excess of the new chunk is 1, insert the chunk in one of the chains and join the chains. If one of e_i and e_r is part of a chunk and the other is part of a necklace, then remove the index in the necklace and insert it into the chunk. Let c_1 and c_2 be the two chains where the two indices were located originally. If the excess of the chunk becomes 0 or 1 then join the chains c_1 and c_2, otherwise let the chunk be a chain by itself by splitting the chain for the chunk. If both e_i and e_r are part of chunks then join them. Check the excess of the joined chunk. If the excess is 2 or more, then make the chunk a chain by its own.

Keep $\mathsf{CHAINLIST}(\alpha)$ updated all the time during the above procedure. Each time a chain is split, joined, or removed, delete the chain from $\mathsf{CHAINLIST}(\alpha)$ (if it is in the list). Each time a new chain is created, e.g. as a result of a split or join, insert the chain into $\mathsf{CHAINLIST}(\alpha)$ if it contain at least one chunk.

Update the global variables $\mathsf{ZS}(\alpha)$ and $\mathsf{NCS}(\alpha)$, respectively, whenever two trees, at the middle or bottom level, respectively, are joined or split.

Every time, during the above described procedure, when two overlapping occurrences with indices e_i and e_j, where $e_i < e_j$, 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 4 *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* $\mathcal{O}(n)$*.*

Proof. The algorithm in Section 5 starts by constructing the suffix tree, $\mathsf{ST}(S)$, for the input string S in time $\mathcal{O}(n \log n)$. The leaf-lists of all n leaves in $\mathsf{ST}(S)$ are created in constant time for each leaf, i.e. in total time $\mathcal{O}(n)$. 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 follows that the total time for processing the internal nodes and edges of $\mathsf{ST}(S)$ is $\mathcal{O}(n)$. In the following we will first give the time for processing the different events. We will then define a potential for each data structure maintained along an edge or node of $\mathsf{ST}(S)$. We will argue that the processing of events of type 1, 2, and 3 release sufficient potential to pay for processing the event, and that a binary merge in an event of type 4 takes time $\mathcal{O}(n_2 \log \frac{n_1+n_2}{n_2})$ and increases the potential by at most the same amount.

When events of type 1 are processed the first step is to find the sorted list of chunks in each chain. For a leaf-list containing m chunks, this is done in time $\mathcal{O}(m)$ since only chains containing at least one chunk are examined and since at least every other element in a chain is a chunk. The next step is to make each chunk a chain by itself by performing at most two split operations per chunk, hence at most $2m$ split operations. The splitting is done from left to right in each chain. Let $|\mathsf{CHAINLIST}(\alpha)| = b$ and let $|c_i|$, for $i = 1, \ldots, b$ be the size of chain i in the list. Let s_{ij} be the size of the left tree resulting from the jth splitting of chain i in $\mathsf{CHAINLIST}(\alpha)$. Then $\sum_{i=1}^{b} |c_i| \leq |\mathsf{LL}(v)|$ and $\sum_{j=1}^{2a_i} s_{ij} \leq |c_i|$ where a_i is the number of chunks in chain i. Hence, $\sum_{i=1}^{b} \sum_{j=1}^{2a_i} s_{ij} \leq |\mathsf{LL}(v)|$. According to Theorem 2 and 3 a split operation takes time $\log(\min\{|T'|, |T''|\})$, where T' and T'' are the resulting trees. The total time of all splittings is at most $\mathcal{O}(\sum_{i=1}^{b} \sum_{j=1}^{2a_i} \log s_{ij}) \leq \mathcal{O}(m \log \frac{|\mathsf{LL}(v)|}{m})$. Finally the chains for the chunks are joined with other chains. Similarly to the split operations, the joining takes time $\mathcal{O}(m \log \frac{|\mathsf{LL}(v)|}{m})$. All other operations, e.g. to recompute the excess and update the global variables, take $\mathcal{O}(m)$. Hence, the total time to process an event of type 1 is $\mathcal{O}(m \log \frac{|\mathsf{LL}(v)|}{m})$.

When an event of type 2 is processed all chunks and necklaces in the same chain are joined into one chain by joining the trees in each chain containing at least one chunk. Since the joining is done from left to right in each chain the total time for all join operations in an event of type 2, like in event of type 1, is $\mathcal{O}(m \log \frac{|\mathsf{LL}(v)|}{m})$, which is also the total time for processing of type event 2.

When an event of type 3 is processed, it is first checked if the two occurrences, that terminate to overlap, belong to the same chain. Let e_1 and e_2 be the indices of the two occurrences and let c be the chain which e_1 belongs to. Denote by $|c|$ the number of occurrences of α in c. Since we have a pointer to e_1, walking up the tree for c to decide if e_2 also belongs to c takes time $\mathcal{O}(\log |c|)$. At the same time it can also be checked if the two indices belong to the same necklace. If the two indices belong to the same chain and/or necklace the chain and/or necklace is split by using at most two split operation on each level, i.e. at most four split operations. According to Theorem 3, a split operation takes time $\mathcal{O}(\log |c|)$. According to Theorem 2, inserting the new chain in the top-level tree takes constant time. Hence, the total time to process an event of type 3 is $\mathcal{O}(\log |c|)$.

For each node v in $\mathsf{ST}(S)$ an event of type 4 is processed, which consists of a sequence of binary mergings as described in Section 5, by viewing $\mathsf{ST}(S)$ as a binary tree. A binary merging is performed by inserting the indices from the smaller of the two subtrees into the data structure for the larger subtree. Denote by $e_1, \ldots, e_{n_1}$ the n_1 indices in the smaller subtree. Denote by $\mathsf{D}(\alpha)$ the data structure for the larger subtree and let n_2 be the number of indices in $\mathsf{D}(\alpha)$. When $e_1, \ldots, e_{n_1}$ are inserted into $\mathsf{D}(\alpha)$ the tree T for $\mathsf{D}(\alpha)$ is split at $e_1, \ldots, e_{n_1}$ into $n_1 + 1$ smaller trees $T_0, \ldots, T_{n_1+1}$. Let s_0 be the number of indices in $\mathsf{D}(\alpha)$ smaller than e_1, let s_i, for $i = 1, \ldots, n_1 - 1$, be the number of indices larger than e_i and smaller than e_{i+1} in $\mathsf{D}(\alpha)$, and let s_{n_1} be the number of indices in $\mathsf{D}(\alpha)$ larger than e_{n_1}. According to Theorems 2 and 3 split takes time $\mathcal{O}(\log \min\{|T'|, |T''|\})$, where T' and T'' are the two resulting trees. It follows that the total time for the splitting at $e_1, \ldots, e_{n_1}$ is $\sum_{i=1}^{n_1+1} \mathcal{O}(\log s_i) = \sum_{i=1}^{n_1+1} \mathcal{O}(\log \frac{n_1+n_2}{n_1}) = \mathcal{O}(n_1 \log \frac{n_1+n_2}{n_1})$. After the splitting the trees and the indices $e_1, \ldots, e_{n_1}$ are joined. The time bound for joining is proportional to the time bound for splitting. All other operations, like updating $\mathsf{NCS}(\alpha)$ and $\mathsf{ZS}(\alpha)$, take time $\mathcal{O}(n_1)$. Since both splitting and joining takes time $\mathcal{O}(n_1 \log \frac{n_1+n_2}{n_1})$ it follows from Lemma 1 that the total time for processing all events of type 4 is $\mathcal{O}(n \log n)$.

We now turn to combine the above time bounds for the different events into an amortization argument, showing that the running time of the algorithm is dominated by the time for all events of type 4, i.e. the running time of the algorithm is $\mathcal{O}(n \log n)$. 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. The data structure $\mathsf{D}(\alpha)$ has potential $\Phi(\mathsf{D}(\alpha))$. Let $\mathcal{C}$ be the set of chains stored in $\mathsf{D}(\alpha)$. For a chain c, let $|c|$ denote the number of occurrences of α in c. We define the potential of $\mathsf{D}(\alpha)$ by the sum

$$\Phi(\mathsf{D}(\alpha)) = \Phi_1(\alpha) + \Phi_2(\alpha) + \sum_{c \in \mathcal{C}} \Phi_3(c) \ ,$$

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. Let k denote the number of chunks in $\mathsf{D}(\alpha)$ and let g denote the number of *green* chunks (defined below) in $\mathsf{D}(\alpha)$. The three potentials Φ_1, Φ_2, and Φ_3, are defined by

$$\begin{aligned} \Phi_1(\alpha) &= 7g \log \frac{|v| \cdot e}{g} \ , \\ \Phi_2(\alpha) &= k \log \frac{|v| \cdot e}{k} \ , \\ \Phi_3(c) &= 2|c| - \log |c| - 2 \ , \end{aligned}$$

with the exceptions that $\Phi_1(\alpha) = 0$ if $g = 0$, and $\Phi_2(\alpha) = 0$ if $k = 0$. Note that the potential of a leaf of $\mathsf{ST}(()S)$ is zero and that Φ_1 and Φ_2 are

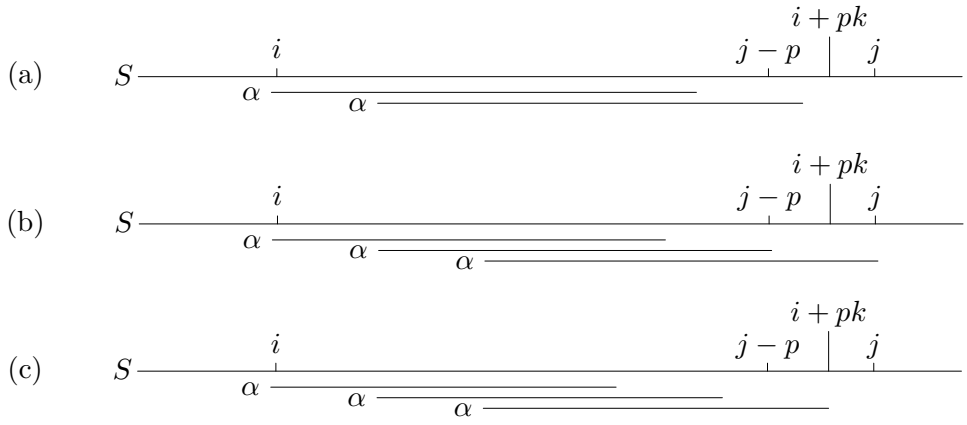


Figure 8: Coloring of chunks while shortening α: (a) red chunk, (b) green chunk, (c) recomputation of excess and green-to-red color change

monotonic increasing for g and k in the range $[0\,..\,|v|]$. For a chunk, with leftmost occurrence of α at position i, consider the substring $S[i\,..\,j]$ with maximal j and $S[i\,..\,j]$ having period p, where $p = p(\alpha)$ is the period of α. Note that the rightmost occurrence at position r in the chunk always covers position $j-p+1$, since otherwise there is an occurrence of α at position $r+p$. We denote the chunk *green* if and only if $|\alpha| \bmod p \leq j-i+1 \bmod p$. Otherwise the chunk is *red*.

The important property of the color of chunks is that when a chunk changes color from red to green while shortening α, then there exists a node in $\mathsf{ST}(S)$ where the merging of the leaf lists increases the number of occurrences in the chunk by exactly one occurrence, i.e. it is sufficient to charge each insertion in a binary merge by the creation of one green chunk. The increase in the potential Φ_1 can then be charged to the processing of events of type 4. When a chunk becomes green in $\mathsf{ST}(S)$, then $|\alpha| \bmod p = j - i + 1 \bmod p$, implying that the last occurrence of α in the chunk is at position $j - |\alpha| + 1$ and covers position j but not $j + 1$. Since $S[j + 1] \neq S[j + 1 - p]$, it follows that the indices of the occurrences of α at positions $j-|\alpha|+1$ and $j-\alpha+1-p$ cannot be contained in the same leaf-list of a child, i.e. the locus of α in $\mathsf{ST}(S)$ is a node v of degree at least two, and the chunk consists of indices from the leaf-lists of two distinct children of v. Figure 8 illustrates the color changes for a chunk while shortening α.

Consider an event of type 1 where the excess of all k chunks is recomputed. This event only occurs when $|a| \bmod p = 0$, i.e. all chunks are green and $g = k$. After the recomputation of the excess all chunks become red. The change of excess of the k chunks in the worst-case implies that the number of chains is reduced by $2k$, because chunks previously having excess ≥ 2 now have excess

≤ 1 and are merged with overlapping necklaces. It follows that $\sum_{c\in\mathcal{C}} \Phi_3(c)$ is increased by at most $2k \log \frac{|v|}{2k} + 4k \leq 6k \log \frac{|v|}{k}$. The change in potential is $\Delta\Phi(\mathsf{D}(\alpha)) \leq -7k \log \frac{|v|\cdot e}{k} + 6k \log \frac{|v|}{k} \leq -k \log \frac{|v|}{k}$, i.e. sufficient potential is released to pay for the $\mathcal{O}(k \log \frac{|v|}{k})$ time for processing the event.

Consider an event of type 2 where all k chunks become necklaces, and where g of the k chunks are green. Converting the k chunks to necklaces does not change the number of chains, since all chunks have excess zero or one. The change in potential is $\Delta\Phi(\mathsf{D}(\alpha)) \leq -k \log \frac{|v|}{k}$, i.e. sufficient potential is released to pay for the $\mathcal{O}(k \log \frac{|v|}{k})$ time for processing the event.

Consider an event 3 where a chain is split into two chains of size n_1 and n_2, where $n_1 \leq n_2$. The change in potential is $\Delta\Phi(\mathsf{D}(\alpha)) = \Delta\Phi_3(\mathsf{D}(\alpha)) \leq (2n_1 - \log n_1 - 2) + (2n_2 - \log n_2 - 2) - (2(n_1 + n_2) - \log(n_1 + n_2) - 2) = \log(n_1 + n_2) - \log n_1 - \log n_2 - 2 \leq \log(2n_2) - \log n_1 - \log n_2 - 2 = -\log n_1 - 1$, i.e. sufficient potential is released to pay for the $\mathcal{O}(\log n_1)$ time for processing the event.

Above we argued that the total time for processing all events of type 4 is $\mathcal{O}(n \log n)$, by showing that inserting the indices from a leaf-list of size n_1 into the data structure for a leaf-list of size n_2 takes time $\mathcal{O}(n_1 \log \frac{n_1+n_2}{n_1})$, where $n_1 \leq n_2$. The merging can at most create n_1 new chunks in the data structure, and increase the number of green chunks by at most n_1. If there are g green chunks before the merging, then there are at most $g + n_1$ green chunks after the merging and the change in the Φ_1 potential for the resulting data structure is

$$\Delta\Phi_1 \leq 7(g + n_1) \log \frac{(n_1 + n_2) \cdot e}{g + n_1} - 7g \log \frac{n_2 \cdot e}{g} \leq 7n_1 \log \frac{(n_1 + n_2) \cdot e}{g + n_1} ,$$

which is $\mathcal{O}\left(n_1 \log \frac{n_1+n_2}{n_1}\right)$.

Similarly, the merging creates at most n_1 new chunks and we get $\Delta\Phi_2 = \mathcal{O}\left(n_1 \log \frac{n_1+n_2}{n_1}\right)$. For $\sum \Phi_3$ we observe that the n_1 insertions first create n_1 singleton chains with Φ_3 potential zero. Each new singleton chain can then be joined with the chains to the left and right of it. If $c_1, \ldots, c_k$ are the chains which are joined into other chains, $k \leq 2n_1$, then the change in the potential $\sum \Phi_3$ for the data structure is

$$\Delta \sum \Phi_3 \leq \sum_{i=1}^{k} (\log |c_i| + 2) \leq 2k + k \log \frac{n_1 + n_2}{k} = \mathcal{O}(n_1 \log \frac{n_1 + n_2}{n_1}) .$$

We conclude that the total amortized time for the binary merging of two leaf-lists of size n_1 and n_2 is $\mathcal{O}(n_1 \log \frac{n_1+n_2}{n_1})$. By Lemma 1 the total amortized time for handling all events of type 4 is $\mathcal{O}(n \log n)$.

The event queue EQ is implemented as an array with n entries, each containing a linked list of the events with a specific priority. Insertions and

deletions can be done in constant time, given that we maintain a pointer to the event to delete. Hence, the total time for the operations in EQ is limited by the total number of events processed by the algorithm and the space used is $\mathcal{O}(n+m)$, where fm is the maximum number of events in EQ at the same time.

To show that the number of events of type 1 is $\mathcal{O}(n)$ we have to show that the excess changes at most once on an edge (u, v) in the suffix tree. Let u be above v in the suffix tree. Assume that the excess changes twice along (u, v). Let $\mathsf{L}(u) = \alpha_u$ and let $\mathsf{L}(v) = \alpha_v$. Say that the excess changes at length $i \cdot p(\alpha_v)$ and $(i-1) \cdot p(\alpha_v)$, where $|\alpha_v| > i \cdot p(\alpha_v) > (i-1) \cdot p(\alpha_v) > |\alpha_u|$. It follows that $\alpha_v[1\,..\,(i-1) \cdot p(\alpha_v)]$ occurs at both positions 1 and $1 + p(\alpha_v)$ in α_v. By considering the rightmost occurence of α_v in S, this means that there has to be an insertion into the leaf-list between v and u, i.e. there is a node between v and u. This is a contradiction and we conclude that there can not be two excess changes along any single edge. The number of events of type 2 is not more than the number of events of type 1, i.e. at most $\mathcal{O}(n)$. The total number of events of type 3 is limited by the number of insertions of indices into the data structure in processing event of type 4. This is shown above to be at most $\mathcal{O}(n \log n)$. At any time during the execution of the algorithm each leaf is involved in at most one event of type 3 as the first of the indices for the overlapping occurrences, hence at most $\mathcal{O}(n)$ events of type 3 are in EQ at the same time. The total number of events of type 4 is equal to the number of internal nodes in the suffix tree, which is at most $\mathcal{O}(n)$. It follows that the event queue EQ will use $\mathcal{O}(n)$ space and insertions and deletions will take time $\mathcal{O}(n \log n)$ in total. □

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 $o(n \log n)$. In *Proc. 29th International*

Colloquium on Automata, Languages, and Programming, volume 2380 of *Lecture Notes in Computer Science*, pages 728–739. 2002.

[6] G. S. Brodal and C. N. S. Pedersen. Finding maximal quasiperiodicities in strings. In *Proc. 11th Combinatorial Pattern Matching (CPM)*, 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] R. C. Lyndon and M. P. Schutzenberger. The equation $a^m = b^n c^p$ in a free group. *Michigan Mathematical Journal*, 9:289–298, 1962.

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

[17] K. Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms.* Springer Verlag, Berlin, 1984.

[18] J. Stoye and D. Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. *Theoretical Computer Science*, 270:843–856, 2002.

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

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