Basic Research in Computer Science

BRICS RS-99-25 Brodal & Pedersen: Finding Maximal Quasiperiodicities in Strings

Finding Maximal Quasiperiodicities in Strings

Gerth Stølting Brodal
Christian N. S. Pedersen

BRICS Report Series **RS-99-25**

ISSN 0909-0878 **September 1999**

See back inner page for a list of recent BRICS Report Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/99/25/`

Finding maximal quasiperiodicities in strings

Gerth Stølting Brodal* Christian N. S. Pedersen*

Abstract

Apostolico and Ehrenfeucht defined the notion of a maximal quasiperiodic substring and gave an algorithm that finds all maximal quasiperiodic substrings in a string of length n in time $O(n \log^2 n)$. In this paper we give an algorithm that finds all maximal quasiperiodic substrings in a string of length n in time $O(n \log n)$ and space $O(n)$. Our algorithm uses the suffix tree as the fundamental data structure combined with efficient methods for merging and performing multiple searches in search trees. Besides finding all maximal quasiperiodic substrings, our algorithm also marks the nodes in the suffix tree that have a superprimitive path-label.

1 Introduction

Characterizing and finding regularities in strings are important problems in many areas of science. In molecular biology repetitive elements in chromosomes determine the likelihood of certain diseases. In probability theory regularities are important in the analysis of stochastic processes. In computer science repetitive elements in strings are important in e.g. data compression, speech recognition, coding, automata and formal language theory.

A widely studied regularity in strings are consecutive occurrences of the same substring. Two consecutive occurrences of the same substring is called an occurrence of a *square* or a *tandem repeat*. This type of regularity was first studied by Thue [27, 28] at the beginning of this century. Thue showed that it is possible to construct arbitrary long strings over any alphabet of more than two characters that contain no squares. Since then a lot of work has been done to develop efficient methods to detect or count squares in strings. Several methods [12, 20, 25] have been presented that determine if a string of length n contains a square in time $O(n)$. Several methods [6, 9, 11, 18, 19, 26] have been presented that find occurrences of squares in a string of length n in time $O(n \log n)$ plus the time it takes to output the detected squares. Recently

*Basic Research In Computer Science (BRICS), Center of the Danish National Research Foundation, Department of Computer Science, University of Aarhus, Ny Munkegade, 8000 Århus C, Denmark. E-mail: {gerth,cstorm}@brics.dk.

two methods [15, 17] have been presented that find a compact representation of all squares in a string of length n in time $O(n)$.

A way to describe the regularity of an entire string in terms of repetitive substrings is the notion of a *periodic* string. Gusfield [14, page 40] defines string S as periodic if it can be constructed by concatenations of a shorter string α. The shortest string from which S can be generated by concatenations is the *period* of S. A string that is not periodic is *primitive*. Some regularities in strings cannot be characterized efficiently using periods or squares. To remedy this, Ehrenfeucht, as referred in [3], suggested the notation of a *quasiperiodic* string. A string S is quasiperiodic if it can be constructed by concatenations and superpositions of a shorter string α. We say that α covers S. Several strings might cover S. The shortest string that covers S is the *quasiperiod* of S. A covering of S implies that S contains a square, so by the result of Thue not all strings are quasiperiodic. A string that is not quasiperiodic is *superprimitive*. Apostolico, Farach and Iliopoulos [5] presented an algorithm that finds the quasiperiod of a given string of length n in time $O(n)$. This algorithm was simplified and made on-line by Breslauer [8]. Moore and Smyth [24] presented an algorithm that finds all substrings that covers a given string of length n in time $O(n)$.

Similar to the period of a string, the quasiperiod of a string describes a global property of the string, but quasiperiods can also be used to characterize substrings. Apostolico and Ehrenfeucht [4] introduced the notion of maximal quasiperiodic substrings of a string. Informally, a quasiperiodic substring γ of S with quasiperiod α is maximal if no extension of γ can be covered by α or αa, where a is the character following γ in S. Apostolico and Ehrenfeucht showed that the maximal quasiperiodic substrings of S correspond to path-labels of certain nodes in the suffix tree of S, and gave an algorithm that finds all maximal quasiperiodic substrings of a string of length n in time $O(n \log^2 n)$ and space $O(n \log n)$. The algorithm is based on a bottom-up traversal of the suffix tree in which maximal quasiperiodic substrings are detected at the nodes in the suffix tree by maintaining various data structures during the traversal. The general structure of the algorithm resembles the structure of the algorithm by Apostolico and Preparata [6] for finding tandem repeats.

In this paper we present an algorithm that finds all maximal quasiperiodic substrings in a string of length n in time $O(n \log n)$ and space $O(n)$. Similar to the algorithm by Apostolico and Ehrenfeucht, our algorithm finds the maximal quasiperiodic substrings in a bottom-up traversal of the suffix tree. The improved time and space bound is a result of using efficient methods for merging and performing multiple searches in search trees, combined with observing that some of the work done, and data stored, by the Apostolico and Ehrenfeucht algorithm is avoidable. The analysis of our algorithm is based on a stronger version of the well known "smaller-half trick" used in the algo-

rithms in [6, 11, 26] for finding tandem repeats. The stronger version of the "smaller-half trick" is hinted at in [22, Exercise 35] and stated in Lemma 6. In [23, Chapter 5] it is used in the analysis of finger searches, and in [9] it is used in the analysis and formulation of an algorithm to find all maximal pairs with bounded gap in a string.

Recently, and independent of our work, Iliopoulos and Mouchard in [16] reported to have an algorithm for finding all maximal quasiperiodic substrings in a string of length n. They stated the running time of the algorithm to be $O(n \log n)$. Their algorithm differs from our algorithm as it does not use the suffix tree as the fundamental data structure, but uses the partitioning technique used by Crochemore [11] combined with several other data structures. Our algorithm together with the algorithm by Iliopoulos and Mouchard show that finding maximal quasiperiodic substrings in a string can be done in two different ways similar to the difference between the algorithms by Crochemore [11] and Apostolico and Preparata [6] for finding tandem repeats.

The rest of this paper is organized as follows. In Section 2 we define the preliminaries used in the rest of the paper. In Section 3 we state and prove properties of quasiperiodic substrings and suffix trees. In Section 4 we state and prove results about efficient merging of and searching in height-balanced trees. In Section 5 we stated our algorithm to find all maximal quasiperiodic substrings in a string. In Section 6 we analyze the running time of our algorithm and in Section 7 we show how the algorithm can be implemented to use linear space.

2 Definitions

In the following we let $S, \alpha, \beta, \gamma \in \Sigma^*$ denote strings over some finite alphabet Σ. We let $|s|$ denote the length of S, $S[i]$ the ith character in S for $1 \leq i \leq |S|$, and $S[i\,..\,j] = S[i]S[i+1]\cdots S[j]$ a substring of S. A string α *occurs* in a string γ at position i if $\alpha = \gamma[i\,..\,i+|\alpha|-1]$. We say that $\gamma[j]$, for all $i \leq j \leq i+|\alpha|-1$, is covered by the occurrence of α at position i.

A string α *covers* a string γ if every position in γ is covered by an occurrence of α. See Figure 1. Note that if α covers γ then α is both a prefix and a suffix of γ. A string is *quasiperiodic* if it can be covered by a shorter string. A string is *superprimitive* if it is not quasiperiodic, that is, if it cannot

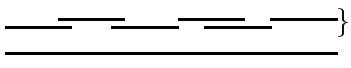

Figure 1: A covering of a string γ by a substring α of γ.

be covered by a shorter string. A superprimitive string α is a quasiperiod of a string γ if α covers γ. In Lemma 1 we show that if α is unique, and α is therefore denoted the *quasiperiod* of γ.

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

For a node v in $T(S)$ we partition $LL(v) = (i_1, i_2, \ldots, i_k)$, $i_j < i_{j+1}$ for $1 \leq j < k$, into a sequence of disjoint subsequences $R_1, R_2, \ldots, R_r$, such that each R_ℓ is a maximal subsequence $i_a, i_{a+1}, \ldots, i_b$, where $i_{j+1} - i_j \leq |L(v)|$ for $a \leq j < b$. Each R_ℓ is denoted a *run* at v and represents a maximal substring of S that can be covered by $L(v)$, i.e. $L(v)$ covers $S[\min R_\ell\,..\,|L(v)| - 1 + \max R_\ell]$, and we say that R_ℓ is a run from $\min R_\ell$ to $|L(v)| - 1 + \max R_\ell$. A run R_ℓ at v is said to *coalesce* at v if R_ℓ contains indices from at least two children of v, i.e. if for no child w of v we have $R_\ell \subseteq LL(w)$. We use $C(v)$ to denote the set of coalescing runs at v.

3 Maximal quasiperiodic substrings

If S is a string and $\gamma = S[i\,..\,j]$ a substring covered by a shorter string $\alpha = S[i\,..\,i+|\alpha|-1]$, then γ is quasiperiodic and we describe it by the triple $(i, j, |\alpha|)$. A triple $(i, j, |\alpha|)$ describes a *maximal quasiperiodic substring* of S, in the following abbreviated *MQS*, if the following requirements are satisfied.

1. $\gamma = S[i\,..\,j]$ is quasiperiodic with quasiperiod α.
2. If α covers $S[i'\,..\,j']$, where $i' \leq i \leq j \leq j'$, then $i' = i$ and $j' = j$.
3. $\alpha S[j+1]$ does not cover $S[i\,..\,j+1]$.

The problem we consider in this paper is for a string S to generate all triples $(i, j, |\alpha|)$ that describe MQSs. This problem was first studied by Apostolico and Ehrenfeucht in [4]. In the following we state important properties of quasiperiodic substrings which are essential to the algorithm to be presented.

Lemma 1 *Every quasiperiodic string γ has a unique quasiperiod α.*

Proof. Assume that γ is cover by two distinct superprimitive strings α and β. Since α and β are prefixes of γ we can without loss of generality assume that α is a proper prefix of β. Since α and β are suffixes of γ, then α is also a proper

suffix of β. Since α and β cover γ, and α is a prefix and suffix of β it follows that α covers β, implying the contradiction that β is not superprimitive. □

Lemma 2 *If γ occurs at position i and j in S, and $1 \leq j - i \leq |\gamma|/2$, then γ is quasiperiodic.*

Proof. Let α be the prefix of γ of length $|\gamma| - (j-i)$, i.e. $\alpha = S[i \,..\, i+|\gamma|-(j-i)-1] = S[j \,..\, i+|\gamma|-1]$. Since $j-i \leq |\gamma|/2$ implies that $i-1+|v|-(j-i) \geq j-1$, we conclude that α covers γ. □

Lemma 3 *If the triple $(i, j, |\alpha|)$ describes a MQS in S, then there exists a non-leaf node in the suffix tree $T(S)$ with path-label α.*

Proof. Assume that α covers the quasiperiodic substring $S[i \,..\, j]$ and that no node in $T(S)$ has path-label α. Since all occurrences of α in S are followed by the same character $a = S[i + |\alpha|]$, αa must cover $S[i \,..\, j+1]$, contradicting the maximality requirement 3. □

Lemma 4 *If γ is a quasiperiodic substring in S with quasiperiod α and u is a non-leaf node in the suffix tree $T(S)$ with path-label γ, then there exists an ancestor node v of u in $T(S)$ with path-label α.*

Proof. Since u is a non-leaf node in $T(S)$ of degree at least two, there exist characters a and b such that both γa and γb occur in S. Since α is a suffix of γ we then have that both αa and αb occur in S, i.e. there exist two suffixes of S having respectively prefix αa and αb, implying that there exists a node v in $T(S)$ with $L(v) = \alpha$. Since α is also a prefix of γ, node v is an ancestor node of u in $T(S)$. □

Lemma 5 *If v is a node in the suffix tree $T(S)$ with a superprimitive path-label α, then the triple $(i, j, |\alpha|)$ describes a MQS in S if and only if there is a run R from i to j that coalesces at v.*

Proof. Let $(i, j, |\alpha|)$ describe a MQS in S and assume that the run $R \in C(v)$ from i and j does not coalesce at v. Then there exists a child v' of v in $T(S)$ such that $R \subseteq LL(v')$. The first symbol along the edge from v to v' is $a = S[i + |\alpha|]$. Every occurrence of α in R is thus followed by a, i.e. αa covers $S[i \,..\, j+1]$. This contradicts the maximality requirement 3 and shows the "if" part of the theorem.

Let R be a coalescing run from i to j at node v, i.e. $L(v) = \alpha$ covers $S[i \,..\, j]$, and let $a = S[j+1]$. To show that $(i, j, |\alpha|)$ describes a MQS in S

it is sufficient to show that αa does not cover $S[i..j+1]$. Since R coalesces at v, there exists a minimal $i'' \in R$ such that αa does not occur in S at position i''. If $i'' = i = \min R$ then αa cannot cover S at position i'' since it by the definition of R cannot occur any position ℓ in S satisfying $i - |\alpha| \leq \ell \leq i$. If $i'' \neq i = \min R$ then αa occurs at $\min R$ and $\max R$, i.e. there exists $i', i''' \in R$, such that $i' < i'' < i'''$, αa occurs at i' and i''' in S, and αa does not occour at any position ℓ in S satisfying $i' < \ell < i'''$. To conclude that $(i, j, |\alpha|)$ describes a MQS we only have to show that $S[i''' - 1]$ is not covered by the occourence of αa at position i', i.e. $i''' - i' > |\alpha| + 1$. By Lemma 2 follows that $i'' - i' > |\alpha|/2$ and $i''' - i'' > |\alpha|/2$, so $i''' - i' \geq |\alpha| + 1$. Now assume that $i''' - i' = |\alpha| + 1$. This implies that $|\alpha|$ is odd and that $i'' - i' = i''' - i'' = (|\alpha| + 1)/2$. Using this we get

$$a = S[i' + |v|] = S[i'' + (|v| - 1)/2] = S[i''' + (|v| - 1)/2] = S[i'' + |v|] \neq a \; .$$

This contradiction shows that $(i, j, |\alpha|)$ describes a MQS and shows the "only if" part of the theorem. □

Theorem 1 *Let v be a non-leaf node in $T(S)$ with path-label α. Since v is a non-leaf node in $T(S)$ there exists $i_1, i_2 \in LL(v)$ such that $S[i_1 + |\alpha|] \neq S[i_2 + |\alpha|]$. The path-label α is quasiperiodic if and only if there exists an ancestor node $u \neq v$ of v in $T(S)$ with path-label β that for $\ell = 1$ or $\ell = 2$ satisfies the following two conditions.*

1. *Both i_ℓ and $i_\ell + |\alpha| - |\beta|$ belong to a coalescing run R at u, and*
2. *for all $i', i'' \in LL(u)$, $|i' - i''| > |\beta|/2$.*

Proof. If α is superprimitive, then no string β covers α, i.e. there exists no node u in $T(S)$ where $C(u)$ includes a run containing both i_ℓ and $i_\ell + |\alpha| - |\beta|$ for $\ell = 1$ or $\ell = 2$. If α is quasiperiodic, then we argue that the quasiperiod β of α satisfies 1 and 2. Since β is superprimitive, 2 is satisfied by Lemma 2. Since β is the quasiperiod of α, we by Lemma 4 have that β is the path-label of a node u in $T(S)$. Since $\beta = S[i_1..i_1 + |\beta| - 1] = S[i_2..i_2 + |\beta| - 1] = S[i_1 + |\alpha| - |\beta|..i_1 + |\alpha| - 1] = S[i_2 + |\alpha| - |\beta|..i_2 + |\alpha| - 1]$ and $S[i_1 + |\alpha|] \neq S[i_2 + |\alpha|]$ then either $S[i_1 + |\alpha|] \neq S[i_1 + |\beta|]$ or $S[i_2 + |\alpha|] \neq S[i_2 + |\beta|]$, which implies that either i_1 and $i_1 + |\alpha| - |\beta|$ are in a coalescing run at u, or i_2 and $i_2 + |\alpha| - |\beta|$ are in a coalescing run at u. □

Theorem 2 *A triple $(i, j, |\alpha|)$ describes a MQS in S if and only if the following three requirements are satisfied*

1. *There exists a non-leaf node v in $T(S)$ with path-label α.*

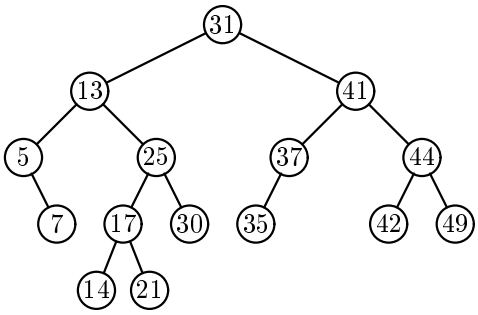

Figure 2: A height-balanced tree.

2. *The path-label α is superprimitive.*
3. *There exists a coalescing run R from i to j at v.*

Proof. The theorem follows directly from the definition of MQS, Lemma 3 and Lemma 5. □

4 Searching and merging height-balanced trees

In this section we consider various operations on height-balanced binary trees, e.g. AVL-trees [1], and present an extension of the well-known "smaller-half trick" which implies a non-trivial bound on the time it takes to perform a sequence of operations on height-balanced binary trees. This bound is essential to the running time of our algorithm for finding maximal quasiperiodic substrings to be presented in the next section.

A height-balanced tree is a binary search tree where each node stores an element from a sorted list, such that for each node v, the elements in the left subtree of v are smaller than the element at v, and the elements in the right subtree of v are larger than the element at v. A height-balanced tree satisfies that for each node v, the heights of the left and right subtree of v differ by at most one. Figure 2 shows a height-balanced tree with 15 elements. A height-balanced tree with n elements has height $O(\log n)$, and element insertions, deletions, and membership queries can be performed in time $O(\log n)$, where updates are based on performing left and right rotations in the tree. We refer to [2] for further details.

For a sorted list $L = (x_1, \ldots, x_n)$ of n distinct elements, and an element x and a value δ, we define the following functions which capture the notation of *predecessors* and *successors* of an element, and the notation of Δ-*predecessors* and Δ-*successors* which in Section 5 will be used to compute the head and

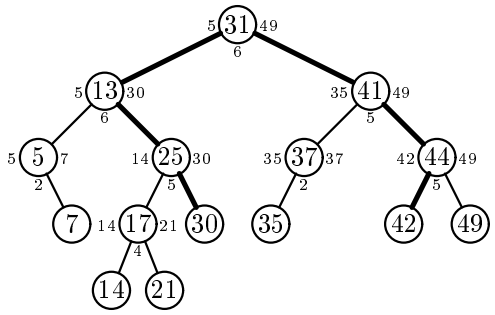

Figure 3: An extended height-balanced tree. Each node with at least one child is annotated with min (left), max (right) and max-gap (bottom). The emphasized path is the search path for Δ-Pred$(T, 4, 42)$

the tail of a coalescing run.

$$\begin{aligned}
\text{pred}(L, x) &= \max\{y \in L \mid y \leq x\}\,, \\
\text{succ}(L, x) &= \min\{y \in L \mid y \geq x\}\,, \\
\text{max-gap}(L) &= \max\{0, x_2 - x_1, x_3 - x_2, \ldots, x_n - x_{n-1}\}\,, \\
\Delta\text{-pred}(L, \delta, x) &= \min\{y \in L \mid y \leq x \ \wedge\ \text{max-gap}(L \cap [y, x]) \leq \delta\}\,, \\
\Delta\text{-succ}(L, \delta, x) &= \max\{y \in L \mid y \geq x \ \wedge\ \text{max-gap}(L \cap [x, y]) \leq \delta\}\,.
\end{aligned}$$

If $L = (5, 7, 13, 14, 17, 21, 25, 30, 31)$, then $\text{pred}(L, 20) = 17$, $\text{succ}(L, 20) = 21$, $\text{max-gap}(L) = 13 - 7 = 6$, $\Delta\text{-pred}(L, 4, 20) = 13$, and $\Delta\text{-succ}(L, 4, 20) = 25$. Note that $\text{pred}(L, x) = \Delta\text{-pred}(L, 0, x)$ and $\text{succ}(L, x) = \Delta\text{-succ}(L, 0, x)$.

In this section we consider an extension of hight-balanced trees where each node v in addition to $\mathsf{key}(v)$, $\mathsf{height}(v)$, $\mathsf{left}(v)$, $\mathsf{right}(v)$, and $\mathsf{parent}(v)$, which respectively stores the element at v, the height of the subtree T_v rooted at v, pointers to the left and right children of v and a pointer to the parent node of v, also stores the following information: $\mathsf{previous}(v)$ and $\mathsf{next}(v)$ are pointers to the nodes which store the immediate predecessor and successor elements of $\mathsf{key}(v)$ in the sorted list, $\mathsf{min}(v)$ and $\mathsf{max}(v)$ are pointers to the nodes storing the smallest and largest elements in the subtree rooted at v, and $\mathsf{max\text{-}gap}(v)$ is the value of max-gap applied to the list of all elements in the subtree T_v rooted at v. The extended height-balanced tree for the tree in Figure 2 is shown in Figure 3 (previous and next pointers are omitted in the figure).

If v has a left child v_1, $\mathsf{min}(v)$ points to $\mathsf{min}(v_1)$. Otherwise $\mathsf{min}(v)$ points to v. Symmetrically, if v has a right child v_2, $\mathsf{max}(v)$ points to $\mathsf{max}(v_2)$. Otherwise $\mathsf{max}(v)$ points to v. If v stores element e and has a left child v_1 and

a right child v_2, then $\mathsf{max\text{-}gap}(v)$ can be computed as

$$\mathsf{max\text{-}gap}(v) = \max\{0, \mathsf{max\text{-}gap}(v_1), \mathsf{max\text{-}gap}(v_2), \\ \mathsf{key}(v) - \mathsf{key}(\mathsf{max}(v_1)), \mathsf{key}(\mathsf{min}(v_2)) - \mathsf{key}(v)\}. \quad (1)$$

If v_1 and/or v_2 do not exist, then the expression is reduced by removing the parts of the expression involving the missing nodes/node. The equation can be used to recompute the information at nodes being rotated when rebalancing a height-balanced search tree. Similar to the function max-gap(L) and the operation $\mathsf{max\text{-}gap}(v)$, we can define and support the function min-gap(L) and the operation $\mathsf{min\text{-}gap}(v)$.

The operations we consider supported for an extended height-balanced tree T are the following, where $e_1, \ldots, e_k$ denotes a sorted list of k distinct elements. The output of the four last operations is a list of k pointers to nodes in T containing the answer to each search key e_i.

- $\mathsf{MultiInsert}(T, e_1, \ldots, e_k)$ inserts (or merges) the k elements into T.
- $\mathsf{MultiPred}(T, e_1, \ldots, e_k)$ for each e_i finds $\mathrm{pred}(T, e_i)$.
- $\mathsf{MultiSucc}(T, e_1, \ldots, e_k)$ for each e_i finds $\mathrm{succ}(T, e_i)$.
- $\mathsf{Multi\text{-}}\Delta\mathsf{\text{-}Pred}(T, \delta, e_1, \ldots, e_k)$ for each e_i finds Δ-pred(T, δ, e_i).
- $\mathsf{Multi\text{-}}\Delta\mathsf{\text{-}Succ}(T, \delta, e_1, \ldots, e_k)$ for each e_i finds Δ-succ(T, δ, e_i).

We merge two height-balanced trees T and T', $|T| \geq |T'|$, by inserting the elements in T' into T, i.e. $\mathsf{MultiInsert}(T, e_1, e_2, \ldots, e_k)$ where $e_1, e_2, \ldots, e_k$ are the elements in T' in sorted order. The following theorem states the running time of the operations.

Theorem 3 *Each of the operations* MultiInsert, MultiPred, MultiSucc, Multi-Δ-Pred, *and* Multi-Δ-Succ *can be performed in time* $O(k \cdot \max\{1, \log(n/k)\})$, *where* n *is the size of the tree and* k *the number elements to be inserted or searched for.*

Proof. If $k \geq n$, then we can in time $O(n)$ convert T to a linear list and answer all multi-queries by performing a linear scan of the list of elements from T and the list $(e_1, \ldots, e_k)$. MultiInsert can be performed in time $O(n)$ by merging the two lists and building a new height-balanced tree. In the following we without loss of generality assume $k \leq n$.

Brown and Tarjan in [10] show how to merge two (standard) height-balanced trees in time $O(k \cdot \max\{1, \log(n/k)\})$, especially their algorithm performs k top-down searches in time $O(k \cdot \max\{1, \log(n/k)\})$. Since a search for an element e either finds the element e or the predecessor/successor of e it follows that MultiPred and MultiSucc can be computed in time $O(k \cdot \max\{1,$

$\log(n/k)\}$) using the previous and next pointers. The implementation of MultiInsert follows from the algorithm of [10] by observing that only the $O(k \cdot \max\{1, \log(n/k)\})$ nodes visited by the merging need to have their associated min, max and max-gap information recomputed due to the inserted elements, and the recomputation can be done in a traversal of these nodes in time $O(k \cdot \max\{1, \log(n/k)\})$ using Equation 1.

We now consider the Multi-Δ-Pred operation. The Multi-Δ-Succ operation is implemented symmetrically to the Multi-Δ-Pred operation, and the details of Multi-Δ-Succ are therefore omitted. The first step of Multi-Δ-Pred is to apply MultiPred, such that for each e_i we find the node v_i with $\mathsf{key}(v_i) = \text{pred}(T, e_i)$. By definition $\Delta\text{-pred}(T, \delta, e_i) = \Delta\text{-pred}(T, \delta, \mathsf{key}(v_i))$. Figure 4 contains code for computing $\Delta\text{-pred}(T, \delta, \mathsf{key}(v_i))$. The procedure $\Delta\text{-pred}(v, \delta)$ finds for a node v in T the node v' with $\mathsf{key}(v') = \Delta\text{-pred}(T, \delta, \mathsf{key}(v))$. The procedure uses the two recursive procedures $\Delta\text{-pred-max}(v, \delta)$ and $\Delta\text{-pred-min}(v, \delta)$ which find nodes v' and v'' satisfying respectively $\mathsf{key}(v') = \Delta\text{-pred}(T_v, \delta, \mathsf{key}(\mathsf{max}(v)))$ and $\mathsf{key}(v'') = \Delta\text{-pred}(T, \delta, \mathsf{key}(\mathsf{min}(v)))$. Note that Δ-pred-max only has search domain T_v. The search $\Delta\text{-pred}(v, \delta)$ basically proceeds in two steps: in the first step a path from v is followed upwards to some ancestor w of v using Δ-pred-min, and in the second step a path is followed from w to the descended of w with key $\Delta\text{-pred}(T, \delta, \mathsf{key}(v))$ using Δ-pred-max. See Figure 3 for a possible search path.

A Δ-predecessor search can be done in time $O(\log n)$, implying that we can find k Δ-predecessors in time $O(k \log n)$. To improve this time bound we apply dynamic programming. Observe that each call to Δ-pred-min corresponds to following a child-parent edge and each call to Δ-pred-max corresponds to following a parent-child edge. By memorizing the results of the calls to Δ-pred-min and Δ-pred-max it follows that each edge is "traversed" in each direction at most once, that all calls to Δ-pred-min and textsfΔ-pred-max correspond to edges in at most k leaf-to-root paths.

From [10, Lemma 6] we have the statement: If T is a height-balanced tree with n nodes, and T' is a subtree of T with at most k leaves, then T' contains $O(k \cdot \max\{1, \log(n/k)\})$ nodes and edges. We conclude that Multi-Δ-Pred takes time $O(k \cdot \max\{1, \log(n/k)\})$, since the time required for the k calls to Δ-Pred is $O(k)$ plus the number of non-memorized recursive calls. □

The "smaller-half trick" states that if each node v in a binary tree supplies a term $O(k)$, where k is the number of leaves in the smallest subtree rooted at a child of v, then the sum over all terms is $O(N \log N)$. The "smaller-half trick" is essential to the running time of several methods for finding tandem repeats [6, 11, 26]. Our method for finding maximal quasiperiodic substrings uses a stronger version of the "smaller-half trick" hinted at in [22, Exercise 35] and stated in Lemma 6. The lemma implies that we at every node in a binary

```
proc Δ-pred(v, δ)
  if left(v) ≠ nil and key(v) − key(max(left(v))) > δ
    return v
  if left(v) = nil or max-gap(left(v)) ≤ δ
    return Δ-pred-min(v, δ)
  return Δ-pred-max(left(v), δ)

proc Δ-pred-max(v, δ)
  if right(v) ≠ nil and (max-gap(right(v)) > δ or key(min(right(v))) − key(v) > δ)
    return Δ-pred-max(right(v), δ))
  if left(v) = nil or (key(v) − key(max(left(v))) > δ
    return v
  return Δ-pred-max(left(v), δ))

proc Δ-pred-min(v, δ)
  if parent(v) = nil
    return min(v)
  if v = left(parent(v))
    return Δ-pred-min(parent(v), δ)
  if key(min(v)) − key(parent(v)) > δ
    return min(v)
  if left(parent(v)) ≠ nil and key(parent(v)) − key(max(left(parent(v)))) > δ
    return parent(v)
  if left(parent(v)) ≠ nil and max-gap(left(parent(v))) > δ
    return Δ-pred-max(left(parent(v)), δ))
  return Δ-pred-min(parent(v), δ)
```

Figure 4: Code for computing the Δ-predecessor of a node in an extended height-balanced tree.

tree with N leaves can perform a fixed number of the operations stated in Theorem 3, with n and k as stated in the lemma, in total time $O(N \log N)$.

Lemma 6 *If each internal node v in a binary tree with N leaves supplies a term $O(k \log(n/k))$, where n is the number of leaves in the subtree rooted at v and $k \leq n/2$ is the number of leaves in the smallest subtree rooted at a child of v, then the sum over all terms is $O(N \log N)$.*

Proof. As the terms are $O(k \log(n/k))$ we can find constants, a and b, such that the terms are upper bounded by $a + bk \log(n/k)$. We will by induction in the number of leaves of the binary tree prove that the sum is upper bounded by $(N-1)a + bN \log N = O(N \log N)$.

If the tree is a leaf then the upper bound holds vacuously. Now assume inductively that the upper bound holds for all trees with at most $N-1$ leaves. 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 these two subtrees, i.e. the sum over all nodes of in the tree except the root, is bounded by $(k-1)a + bk\log k + ((N-k)-1)a + b(N-k)\log(N-k)$. The the entire sum is thus bounded by

$$\begin{aligned} a + bk\log(N/k) + (k-1)a + bk\log k + ((N-k)-1)a + b(N-k)\log(N-k) \\ &= (N-1)a + bk\log N + b(N-k)\log(N-k) \\ &< (N-1)a + bk\log N + b(N-k)\log N \\ &= (N-1)a + bN\log N \end{aligned}$$

which proves the lemma. □

5 Algorithm

The algorithm to find all maximal quasiperiodic substrings in a string S first constructs the suffix tree $T(S)$ of S in time $O(n)$ using any existing suffix tree construction algorithm, e.g. [13, 21, 29, 30], and then processes $T(S)$ in two phases. Each phase involves one or more traversals of $T(S)$. In the first phase the algorithm identifies all nodes of $T(S)$ with a superprimitive path-label. In the second phase the algorithm reports the maximal quasiperiodic substrings in S. This is done by reporting the coalescing runs at the nodes which in the first phase were identified to have superprimitive path-labels.

To identify nodes with superprimitive path-labels we apply the concepts of *questions*, *characteristic occurrences* of a path-label, and *sentinels* of a node. Let v be a non-leaf node in $T(S)$ and $u \neq v$ an ancestor node of v in $T(S)$. Let v_1 and v_2 be the two leftmost children of v, and $i_1 = \min(LL(v_1))$ and $i_2 = \min(LL(v_2))$. A *question posed to* u is a triple (i, j, v) where $i \in LL(v) \subset LL(u)$ and $j = i + |L(v)| - |L(u)| \in LL(u)$, and the answer to the question is true if and only if i and j are in the same coalescing run at u. We define the two occurrences of $L(v)$ at positions i_1 and i_2 to be the *characteristic occurrences* of $L(v)$, and define the *sentinels* $\hat{v}_1$ and $\hat{v}_2$ of v as the positions immediately after the two characteristic occurrences of $L(v)$, i.e. $\hat{v}_1 = i_1 + |L(v)|$ and $\hat{v}_2 = i_2 + |L(v)|$. Since i_1 and i_2 are indices in leaf-lists of two distinct children of v, we have $S[\hat{v}_1] \neq S[\hat{v}_2]$. In the following we let $SL(v)$ be the list of the sentinels of the nodes in the subtree rooted at v in $T(S)$. Since there are two sentinels for each non-leaf node $|SL(v)| \leq 2|LL(v)| - 2$.

Theorem 1 implies the following technical lemma which forms the basis for detecting nodes with superprimitive path-labels in $T(S)$.

Lemma 7 *The path-label $L(v)$ is quasiperiodic if and only if there exists a sentinel $\hat{v}$ of v, and an ancestor w of v (possibly $w = v$) for which there exists $j \in LL(w) \cap]\hat{v} - 2 \cdot \text{min-gap}(LL(w))\,;\,\hat{v}[$ such that $(\hat{v} - |L(v)|, j, v)$ is a question that can be posed and answered successfully at an ancestor node $u \neq v$ of w (possibly $u = w$) with $|L(u)| = \hat{v} - j$ and $\text{min-gap}(LL(u)) > |L(u)|/2$.*

Proof. If there exists a question $(\hat{v} - |L(v)|, \hat{v} - |L(u)|, v)$ that can be answered successfully at u, then $\hat{v} - |L(v)|$ and $\hat{v} - |L(u)|$ are in the same run at u, i.e. $L(u)$ covers $L(v)$ and $L(v)$ is quasiperiodic.

If $L(v)$ is quasiperiodic, we have from Theorem 1 that there for $i_\ell = \hat{v}_\ell - |L(v)|$, where $\ell = 1$ or $\ell = 2$, exists an ancestor node $u \neq v$ of v where both i_ℓ and $i_\ell + |L(v)| - |L(u)|$ belong to a coalescing run at u and $\text{min-gap}(LL(u)) > |L(u)|/2$. The lemma follows by letting $w = u$ and $j = \hat{v}_\ell - |L(u)|$. □

Since j and $\hat{v}$ uniquely determines the question $(\hat{v} - |L(v)|, j, v)$, it follows that in order to decide the superprimitivity of all nodes it is sufficient for each node w to find all pairs $(\hat{v}, j)$ where $\hat{v} \in SL(w)$ and $j \in LL(w) \cap]\hat{v} - 2 \cdot \text{min-gap}(LL(w))\,;\,\hat{v}[$, or equivalently $j \in LL(w)$ and $\hat{v} \in SL(w) \cap]j\,;\,j + 2 \cdot \text{min-gap}(LL(w))[$. Furthermore, if $\hat{v}$ and j result in a question at w, but $j \in LL(w')$ and $\hat{v} \in SL(w')$ for some child w' of w, then $\hat{v}$ and j result in the same question at w' since $\text{min-gap}(LL(w')) \geq \text{min-gap}(LL(w))$, i.e. we only need to find all pairs $(\hat{v}, j)$ at w where $\hat{v}$ and j come from two distinct children of w. We can now state the details of the algorithm.

Phase I – Marking nodes with quasiperiodic path-labels

In Phase I we mark all nodes in $T(S)$ that have a quasiperiodic path-label by performing three traversals of $T(S)$. We first make a depth-first traversal of $T(S)$ where we for each node v compute $\text{min-gap}(LL(v))$. We do this by constructing for each node v a search tree $T_{LL}(v)$ that stores $LL(v)$ and supports the operations in Section 4. In particular the root of $T_{LL}(v)$ should store the value $\mathsf{min\text{-}gap}(T_{LL}(v))$ to be assigned to v. If v is a leaf, $T_{LL}(v)$ only contains the index annotated to v. If v is an internal node, we construct $T_{LL}(v)$ by merging the T_{LL} trees of the children of v from left-to-right when these have been computed. If the children of v are $v_1, \ldots, v_k$ we merge $T_{LL}(v_1), \ldots, T_{LL}(v_{i+1})$ by performing a binary merge of $T_{LL}(v_{i+1})$ with the result of merging $T_{LL}(v_1), \ldots, T_{LL}(v_i)$. As a side effect of computing $T_{LL}(v)$ the T_{LL} trees of the children of v are destroyed. We pose and answer questions in two traversals of $T(S)$ explained below as Step 1 and Step 2. For each node v we let $Q(v)$ contain the list of questions posed at v. Inititially $Q(v)$ is empty.

Step 1 (Generating questions) In this step we perform a depth-first traversal of $T(S)$. At each node v we construct search trees $T_{LL}(v)$ and $T_{SL}(v)$ which store respectively $LL(v)$ and $SL(v)$ and support the operations mentioned in Section 4. For a non-leaf node v with leftmost children v_1 and v_2, we compute the sentinels of v as $\hat{v}_1 = \mathsf{min}(T_{LL}(v_1))$ and $\hat{v}_2 = \mathsf{min}(T_{LL}(v_2))$. The T_{LL} trees need to be recomputed since these are destroyed in the first traversal of $T(S)$. The computation of $T_{SL}(v)$ is done similarly to the computation of $T_{LL}(v)$ by merging the T_{SL} lists of the children of v from left-to-right, except that after the merging the T_{SL} trees of the children we also need to insert the two sentinels $\hat{v}_1$ and $\hat{v}_2$ in $T_{SL}(v)$.

We visit node v, and call it the *current node*, when the T_{LL} and T_{SL} trees at the children of v are available. During the depth-first traversal we maintain an array depth such that $\mathsf{depth}(k)$ is a reference to the node u on the path from the current node to the root with $|L(u)| = k$ if such a node exists. Otherwise $\mathsf{depth}(k)$ has the value undef. We maintain depth by setting $\mathsf{depth}(|L(u)|)$ to u when we arrive at u from its parent, and setting $\mathsf{depth}(|L(u)|)$ to undef when we return from u to its parent.

When v is the current node we have from Lemma 7 that it is sufficient to generate questions for pairs $(\hat{w}, j)$ where $\hat{w}$ and j come from two different children of v. We do this while merging the T_{LL} and T_{SL} trees of the children. Let the children of v be $v_1, \ldots, v_k$. Assume $LL_i = LL(v_1) \cup \cdots \cup LL(v_i)$ and $SL_i = SL(v_1) \cup \cdots \cup SL(v_i)$ has been computed as T_{LL_i} and T_{SL_i} and we are in the process of computing LL_{i+1} and SL_{i+1}. The questions we need to generate while computing LL_{i+1} and SL_{i+1} are those where $j \in LL_i$ and $\hat{w} \in SL(v_{i+1})$ or $j \in LL(v_{i+1})$ and $\hat{w} \in SL_i$. Assume $j \in T_{LL}$ and $\hat{w} \in T_{SL}$, where either $T_{LL} = T_{LL_i}$ and $T_{SL} = T_{SL}(v_{i+1})$ or $T_{LL} = T_{LL}(v_{i+1})$ and $T_{SL} = T_{SL_i}$. There are two cases. If $|T_{LL}| \leq |T_{SL}|$ we locate each $j \in T_{LL}$ in T_{SL} by performing a $\mathsf{MultiSucc}$ operation. Using the next pointers we can then for each j report those $\hat{w} \in T_{SL}$ where $\hat{w} \in \,]j\,;\, j + \text{min-gap}(v)[$. If $|T_{LL}| > |T_{SL}|$ we locate each $\hat{w} \in T_{SL}$ in T_{LL} by performing a $\mathsf{MultiPred}$ operation. Using the $\mathsf{previous}$ pointers we can then for each $\hat{w}$ report those $j \in T_{SL}$ where $j \in \,]\hat{w} - \text{min-gap}(v)\,;\, \hat{w}[$. The two sentinels $\hat{v}_1$ and $\hat{v}_2$ of v are handled similarly to the later case by performing two searches in $T_{LL}(v)$ and using the $\mathsf{previous}$ pointers to generate the required pairs involving the sentinels $\hat{v}_1$ and $\hat{v}_2$ of v.

For a pair $(\hat{w}, j)$ generated at the current node v we generate a question $(\hat{w} - |L(w)|, j, w)$ about descendent w of v with sentinel $\hat{w}$ and pose the question at ancestor $u = \mathsf{depth}(\hat{w} - j)$ by inserting $(\hat{w} - |L(w)|, j, w)$ into $Q(u)$. If u does not exists, i.e. $\mathsf{depth}(\hat{w} - j)$ is undef, or $\mathsf{min\text{-}gap}(LL(u)) \leq |L(u)|/2$ then no question is posed.

Step 2 (Answering questions) Let $Q(v)$ be the set of questions posed at node v in Step 1. If there is a coalescing run R in $C(v)$ and a question (i, j, w) in $Q(v)$ such that $\min R \leq i < j \leq \max R$, then i and j are in the same coalescing run at v and we mark node w as having a quasiperiodic path-label.

We identify each coalescing run R in $C(v)$ by the tuple $(\min R, \max R)$. We *answer* question (i, j, w) in $Q(v)$ by deciding if there is a run $(\min R, \max R)$ in $C(v)$ such that $\min R \leq i < j \leq \max R$. If the questions (i, j, w) in $Q(v)$ and runs $(\min R, \max R)$ in $C(v)$ are sorted lexicographically, we can answer all questions by a linear scan through $Q(v)$ and $C(v)$. In the following we describe how to generate $C(v)$ in sorted order and how to sort $Q(v)$.

Constructing coalecsing runs The coalescing runs are generated in a traversal of $T(S)$. At each node v we construct $T_{LL}(v)$ storing $LL(v)$. We construct $T_{LL}(v)$ by merging the T_{LL} trees of the children of v from left-to-right. A coalescing run R in $LL(v)$ contains an index from at least two distinct children of v, i.e. there are indices $i' \in LL(v_1)$ and $i'' \in LL(v_2)$ in R for two distinct children v_1 and v_2 of v such that $i' < i''$ are neighbors in $LL(v)$ and $i'' - i' \leq |L(v)|$. We say that i' is a *seed* of R. We identify R by the tuple $(\min R, \max R)$. We have $\min R = \Delta\text{-pred}(LL(v), |L(v)|, i')$ and $\max R = \Delta\text{-succ}(LL(v), |L(v)|, i')$.

To construct $C(v)$ we collect seeds $i_{r_1}, i_{r_2}, \ldots, i_{r_k}$ of every coalescing run in $LL(v)$ in sorted order. This done by checking while merging the T_{LL} trees of the children of v if an index gets a new neighbor in which case the index can be identified as a seed. Since each insertion at most generates two seeds we can collect all seeds into a sorted list while performing the merging. From the seeds we can compute the first and last index of the coalesing runs by doing $\mathsf{Multi\text{-}\Delta\text{-}Pred}(T_{LL}(v), |L(v)|, i_{r_1}, i_{r_2}, \ldots, i_{r_k})$ and $\mathsf{Multi\text{-}\Delta\text{-}Succ}(T_{LL}(v), |L(v)|, i_{r_1}, i_{r_2}, \ldots, i_{r_k})$. Since we might have collected several seeds of the same run, the list of coalescing runs $R_1, R_2, \ldots, R_k$ might contain duplets which can be removed by reading through the list once. Since the seeds is collected in sorted order, the resulting list of runs is also sorted.

Sorting the questions We collect the elements in $Q(v)$ for every node v in $T(S)$ into a single list Q that contains all question (i, j, w) posed at nodes in $T(S)$. We annotate every element in Q with the node v it was collected from. By construction every question (i, j, w) posed at a node in $T(S)$ satisfies that $0 \leq i < j < n$. We can thus sort the elements in Q lexicographically with respect to i and j using radix sort. After sorting the elements in Q we distribute the questions back to the proper nodes in sorted order by a linear scan through Q.

Phase II – Reporting maximal quasiperiodic substrings

After Phase I all nodes that have a quasiperiodic path-label are marked, i.e. all unmarked nodes are nodes that have a superprimitive path-label. By Theorem 2 we report all maximal quasiperiodic substrings by reporting the coalescing runs at every node that has a superprimitive path-label. In a traversal of the marked suffix tree we as in Phase I construct $C(v)$ at every unmarked node and report for every R in $C(v)$ the triple $(\min R, \max R, |L(v)|)$ that identifies the corresponding maximal quasiperiodic substring.

6 Running time

In every phase of the algorithm we traverse the suffix tree and construct at each node v search trees that stores $LL(v)$ and/or $SL(v)$. At every node v we construct various lists by considering the children of v from left-to-right and perform a constant number of the operations in Theorem 3. Since the overall merging of information in $T(S)$ is done by binary merging we by Lemma 6 have that this amounts to time $O(n \log n)$ in total. To generate and answer questions we use time proportional to the total number of questions generated. Lemma 8 state that the number of questions is bounded by $O(n \log n)$. We conclude that the running time of the algorithm is $O(n \log n)$.

Lemma 8 *At most $O(n \log n)$ questions are generated.*

Proof. We prove that each of the $2n$ sentinels can at most result in the generation of $O(\log n)$ questions. Consider a sentinel $\hat{w}$ of node w and assume that it generates a question $(\hat{w} - |L(w)|, j, w)$ at node v. Since $\hat{w} - j < 2 \cdot \text{min-gap}(v)$, j is either $\text{pred}(LL(v), \hat{w} - 1)$ (a question of Type A) or the left neighbor of $\text{pred}(LL(v), \hat{w} - 1)$ in $LL(v)$ (a question of Type B). For $\hat{w}$ we first consider all indices resulting in questions of Type A along the path from w to the root. Note that this is an increasing sequence of indices. We now show that the distance of $\hat{w}$ to the indices is geometrically decreasing, i.e. there are at most $O(\log n)$ questions generated of Type A. Let j and j' be two consecutive indices resulting in questions of Type A at node v and at an ancestor node u of v. Since $j < j' < \hat{w}$ and $j' - j \geq \text{min-gap}(u)$ and $\hat{w} - j' < 2 \cdot \text{min-gap}(u)$, we have that $\hat{w} - j' < \frac{2}{3}(\hat{w} - j)$. Similarly we can bound the number of questions generated of Type B for sentinel $\hat{w}$ by $O(\log n)$. □

7 Achieving linear space

Storing the suffix tree $T(S)$ uses space $O(n)$. During a traversal of the suffix tree we construct search trees as explained. Since no element, index or sentinel,

at any time is stored in more than a constant number of search trees, storing the search trees uses space $O(n)$. Unfortunately, storing the sets $C(v)$ and $Q(v)$ of coalescing runs and questions at every node v in the suffix tree uses space $O(n \log n)$. To reduce the space consumption we must thus avoid to store $C(v)$ and $Q(v)$ at all nodes simultaneously. The trick is to modify Phase I to alternate between generating and answering questions.

We observe that generating questions and coalescing runs (Step 1 and the first part of Step 2) can be done in a single traversal of the suffix tree. This traversal is Part 1 of Phase I. Answering questions (the last part of Step 1) is Part 2 of Phase I. To reduce the space used by the algorithm to $O(n)$ we modify Phase I to alternate in rounds between Part 1 (generating questions and coalescing runs) and Part 2 (answering questions).

We say that node v is *ready* if $C(v)$ is available and all questions from it has been generated, i.e. Part 1 has been performed on it. If node v is ready then all nodes in its subtree are ready. Since all questions to node v are generated at nodes in its subtree, this implies that $Q(v)$ is also available. By definition no coalescing runs are stored at non-ready nodes and Lemma 9 states that only $O(n)$ questions are stored at non-ready nodes. In a round we produce ready nodes (perform Part 1) until the number of questions plus coalescing runs stored at nodes readied in the round exceed n, we then answer the questions (perform Part 2) at nodes readied in the round. After a round we dispose questions and coalescing runs stored at nodes readied in the round. We continue until all nodes in the suffix tree have been visited.

Lemma 9 *There are at most $O(n)$ questions stored at non-ready nodes.*

Proof. Let v be a node in $T(S)$ such that all nodes on the path from v to the root are non-ready. Consider a sentinel $\hat{w}$ corresponding to a node in the subtree rooted at v. Assume that three questions $(\hat{w} - |L(w)|, j', w)$, $(\hat{w} - |L(w)|, j'', w)$ and $(\hat{w} - |L(w)|, j''', w)$ where $j' < j'' < j'''$, because of $\hat{w}$ have been posed to ancestors of v, i.e. non-ready nodes. Consider node $u = \mathsf{depth}(\hat{w} - j')$. Since question $(\hat{w} - |L(w)|, j', w)$ is posed at u, min-gap$(LL(u)) > |L(u)|/2$. Since $j', j'', j''' \in LL(u)$ and $j''' - j' \leq \hat{w} - j' = |L(u)|$, min-gap$(LL(u)) \leq \min\{j'' - j', j''' - j''\} \leq |L(u)|/2$. This contradicts that min-gap$(LL(u)) > |L(u)|/2$ and shows that each sentinel has generated at most two questions to non-ready nodes. The lemma follows because there are at most $2n$ sentinels in total. □

Alternating between Part 1 and Part 2 clearly results in generating and answering the same questions as if Part 1 and Part 2 were performed without alternation. The correctness of the algorithm is thus unaffected by the modification of Phase I. Now consider the running time. The running time of a round can be divided into time spent on readying nodes (Part 1) and

time spent on answering questions (Part 2). The total time spent on readying nodes is clearly unaffected by the alternation. To conclude the same for the total time spent on answering questions, we must argue that the time spend on sorting the posed questions in each round is proportional to the time otherwise spend in the round. The crucial observation is that each round takes time $\Omega(n)$ for posing questions and identifying coalescing runs, implying that the $O(n)$ term in each radix sorting is neglectable. We conclude that the running time is unaffected by the modification of Phase I. Finally consider the space used by the modified algorithm. Besides storing the suffix tree and the search trees which uses space $O(n)$, it only stores $O(n)$ questions and coalescing runs at nodes readied in the current round (by construction of a round) and $O(n)$ questions at non-ready nodes (by Lemma 9). In summary we have the following theorem.

Theorem 4 *All maximal quasiperiodic substrings of a string of length n can be found in time $O(n \log n)$ and space $O(n)$.*

8 Conclusion

We have presented an algorithm that finds all maximal quasiperiodic substrings of a string of length n in time $O(n \log n)$ and space $O(n)$. Besides improving on a previous algorithm by Apostolico and Ehrenfeucht, the algorithm demonstrates the usefulness of suffix trees combined with efficient methods for merging and performing multiple searches in search trees. We believe that the techniques presented in this paper could also be useful in improving the running time of the algorithm for the string statistic problem presented by Apostolico and Preparata [7] to $O(n \log n)$.

References

[1] G. M. Adel'son-Vel'skii and Y. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962. English translation in *Soviet Math. Dokl.*, 3:1259–1262.

[2] A. V. Aho, J. E. Hopcraft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison–Wesley, Reading, Massachusetts, 1974.

[3] A. Apostolico and D. Breslauer. Of periods, quasiperiods, repetitions and covers. In *A selection of essays in honor of A. Ehrenfeucht*, volume 1261 of *Lecture Notes in Computer Science*. Springer, 1997.

[4] A. Apostolico and A. Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theoretical Computer Science*, 119:247–265, 1993.

[5] A. Apostolico, M. Farach, and C. S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39:17–20, 1991.

[6] A. Apostolico and F. P. Preparata. Optimal off-line detection of repetitions in a string. *Theoretical Computer Science*, 22:297–315, 1983.

[7] A. Apostolico and F. P. Preparata. Data structures and algorithms for the string statistics problem. *Algorithmica*, 15:481–494, 1996.

[8] D. Breslauer. An on-line string superprimitivity test. *Information Processing Letters*, 44:345–347, 1992.

[9] G. S. Brodal, R. B. Lyngsø, C. N. S. Pedersen, and J. Stoye. Finding maximal pairs with bounded gap. In *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1645 of *Lecture Notes in Computer Science*, pages 134–149, 1999.

[10] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *Journal of the ACM*, 26(2):211–226, 1979.

[11] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981.

[12] M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.

[13] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 137–143, 1997.

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

[15] D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. Technical Report CSE-98-4, Department of Computer Science, UC Davis, 1998.

[16] C. S. Iliopoulos and L. Mouchard. Quasiperiodicity: from detection to normal form. *Journal of Automata, Languages and Combinatorics*, 1999. To appear.

[17] R. Kolpakov and G. Kucherov. Maximal repetitions in words or how to find all squares in linear time. Technical Report 98-R-227, LORIA, 1998.

[18] S. R. Kosaraju. Computation of squares in a string. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 807 of *Lecture Notes in Computer Science*, pages 146–150, 1994.

[19] M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5:422–432, 1984.

[20] M. G. Main and R. J. Lorentz. Linear time recognition of squarefree strings. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pages 271–278. Springer, Berlin, 1985.

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

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

[23] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999. To appear. See http://www.mpi-sb.mpg.de/~mehlhorn/LEDAbook.html.

[24] D. Moore and W. F. Smyth. Computing the covers of a string in linear time. In *Proceedings of the 5th Annual Symposium on Discrete Algorithms (SODA)*, pages 511–515, 1994.

[25] M. Rabin. Discovering repetitions in strings. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pages 279–288. Springer, Berlin, 1985.

[26] J. Stoye and D. Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. In *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1448 of *Lecture Notes in Computer Science*, pages 140–152, 1998.

[27] A. Thue. Über unendliche Zeichenreihen. *Skrifter udgivne af Videnskabs-Selskabet i Christiania, Mathematisk-Naturvidenskabelig Klasse*, 7:1–22, 1906.

[28] A. Thue. Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen. *Skrifter udgivne af Videnskabs-Selskabet i Christiania, Mathematisk-Naturvidenskabelig Klasse*, 1:1–67, 1912.

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

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

Recent BRICS Report Series Publications

RS-99-25 Gerth Stølting Brodal and Christian N. S. Pedersen. *Finding Maximal Quasiperiodicities in Strings*. September 1999. 20 pp.

RS-99-24 Luca Aceto, Willem Jan Fokkink, and Chris Verhoef. *Conservative Extension in Structural Operational Semantics*. September 1999. 23 pp. To appear in the *Bulletin of the EATCS*.

RS-99-23 Olivier Danvy, Belmina Dzafic, and Frank Pfenning. *On proving syntactic properties of CPS programs*. August 1999. 14 pp. To appear in Gordon and Pitts, editors, *3rd Workshop on Higher Order Operational Techniques in Semantics*, HOOTS '99 Proceedings, ENTCS, 1999.

RS-99-22 Luca Aceto, Zoltán Ésik, and Anna Ingólfsdóttir. *On the Two-Variable Fragment of the Equational Theory of the Max-Sum Algebra of the Natural Numbers*. August 1999. 22 pp.

RS-99-21 Olivier Danvy. *An Extensional Characterization of Lambda-Lifting and Lambda-Dropping*. August 1999. 13 pp. Extended version of an article to appear in *Fourth Fuji International Symposium on Functional and Logic Programming*, FLOPS '99 Proceedings (Tsukuba, Japan, November 11–13, 1999). This report supersedes the earlier report BRICS RS-98-2.

RS-99-20 Ulrich Kohlenbach. *A Note on Spector's Quantifier-Free Rule of Extensionality*. August 1999. 5 pp. To appear in *Archive for Mathematical Logic*.

RS-99-19 Marcin Jurdziński and Mogens Nielsen. *Hereditary History Preserving Bisimilarity is Undecidable*. June 1999. 18 pp.

RS-99-18 M. Oliver Möller and Harald Rueß. *Solving Bit-Vector Equations of Fixed and Non-Fixed Size*. June 1999. 18 pp. Revised version of an article appearing under the title *Solving Bit-Vector Equations* in Gopalakrishnan and Windley, editors, *Formal Methods in Computer-Aided Design: Second International Conference*, FMCAD '98 Proceedings, LNCS 1522, 1998, pages 36–48.

RS-99-17 Andrzej Filinski. *A Semantic Account of Type-Directed Partial Evaluation*. June 1999. To appear in Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, PPDP99 '99 Proceedings, LNCS, 1999.