Efficient Counting of Square Substrings in a Tree*

Tomasz Kociumaka¹, Jakub Pachocki¹, Jakub Radoszewski¹, Wojciech Rytter¹,², and Tomasz Walen³,¹

¹Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Poland
²Faculty of Mathematics and Computer Science, Copernicus University, Toruń, Poland
³Laboratory of Bioinformatics and Protein Engineering, International Institute of Molecular and Cell Biology in Warsaw, Poland

Abstract

We give an algorithm which in \( O(n \log^2 n) \) time counts all distinct squares in a labeled tree. There are two main obstacles to overcome. The first one is that the number of distinct squares in a tree is \( \Omega(n^{4/3}) \) (see Crochemore et al., CPM 2012), which differs substantially from the case of classical strings for which there are only linearly many distinct squares. We overcome this obstacle by using a compact representation of all squares (based on maximal cyclic shifts) which requires only \( O(n \log n) \) space. The second obstacle is lack of adequate algorithmic tools for labeled trees, consequently we design several novel tools, this is the most complex part of the paper. In particular we extend to trees Imre Simon’s compact representations of the failure table in pattern matching machines.

1 Introduction

Repetitions play an important role in combinatorics on words with particular applications in pattern matching, text compression, computational biology etc. For a survey on known results related to repetitions in words and their applications see [2]. The basic type of a repetition are squares: strings of the form \( ww \). Here we consider square substrings corresponding to simple paths in labeled unrooted trees. Squares in trees and graphs have already been considered e.g. in [3, 4]. There have also been results on squares in partial words [5] and squares in the context of games [6].

Recently it has been shown that a tree with \( n \) nodes can contain \( \Theta(n^{4/3}) \) distinct squares, see [7], while the number of distinct squares in a string of length \( n \) does not exceed \( 2n - \Theta(\log n) \), as shown in [8, 9, 10]. This paper can be viewed as an algorithmic continuation of [7].

Enumerating squares in ordinary strings is already a difficult problem, despite the linear upper bound on their number. Complex \( O(n) \) time solutions to this problem using suffix trees [11] and runs [12] are known. Two notions that we introduce in this paper (semiruns and packages of cyclically equivalent squares) are in a sense an extension of the techniques used in [12].

Assume we have a tree \( T \) with \( n \) nodes whose edges are labeled with symbols from an integer alphabet \( \Sigma \). We assume that \( \Sigma \) is polynomially bounded in terms of \( n \), i.e. \( \Sigma \subset \{0, \ldots, n^C\} \) for some positive integer constant \( C \). If \( u \) and \( v \) are two nodes of \( T \), then let \( val(u, v) \) denote the sequence of labels of edges on the path from \( u \) to \( v \). We call \( val(u, v) \) a substring of \( T \). (Note that a substring is a string, not a path.) Denote by \( sq(T) \) the set of different square substrings in \( T \). The main problem we consider is as follows:

Problem

Input: A labeled tree \( T \).

*A preliminary version of this paper appeared at ISAAC 2012 [1].
Output: The number of distinct square substrings in $T$.

Example: For the tree in Fig. 1 we have $|\text{sq}(T)| = 5$.

![Figure 1](image)

Figure 1: This tree contains the following squares: $aa$, $aaaa$, $abab$, $baba$, $bb$. We have here $|\text{sq}(T)| = 5$. Note that the squares $abab$ and $baba$ correspond to the same path, one is read from left to right and the other is read from right to left.

Our result: We compute $|\text{sq}(T)|$ in $O(n \log^2 n)$ time.

In the same time complexity we provide a compact representation of the set of all distinct squares in $T$. The representation consists of $O(n \log n)$ packages, each package stores a pair of nodes $x, y$ that represents the maximum cyclic rotation $u = \text{val}(x, y)$ of a square half and a cyclic interval $I$ such that for all $i \in I$ the cyclic rotation of $u$ by $i$ letters is also a square half (see Fig. 4 for an example).

The structure of the algorithm:

1. We reduce the problem to finding a compact representation of all squares anchored at a given node $r$ of the tree. Afterwards we consider trees rooted at specific node $r$.

2. For a given root $r$ we compute a set of paths, called semiruns, which contain all squares anchored at $r$.

3. We reorganize the data related to semiruns to get an unambiguous representation of squares in terms of cyclic rotations. This unambiguity allows efficient counting of squares.

The hardest and the most interesting part of the paper is efficient computation of several basic tables. We structure the paper in such a way that this part is moved after the presentation of the main algorithm (which is described in Section 4). Before that we present combinatorial tools related to strings and labeled trees which are the base of the algorithm design.

In the last section we present a linear time algorithm for counting squares in a special family of trees called combs. The trees from this family turn out to maximize the asymptotic number of square substring [7].

2 Combinatorial Tools for Squares in Trees

Centroid decomposition The centroid decomposition enables to consider paths going through the root in rooted trees instead of arbitrary paths in an unrooted tree. Let $T$ be an unrooted tree of $n$ nodes. Let $T_1, T_2, \ldots, T_k$ be the connected components obtained after removing a node $r$ from $T$. The node $r$ is called a centroid of $T$ if $|T_i| \leq n/2$ for all $T_i$. The centroid decomposition of $T$, $\text{CDecomp}(T)$, is defined recursively:

$$\text{CDecomp}(T) = \{(T, r)\} \cup \bigcup_{i=1}^{k} \text{CDecomp}(T_i).$$

Every tree has a centroid, see [13], and a centroid of a tree can be computed in $O(n)$ time. The recursive definition of $\text{CDecomp}(T)$ implies a bound on its total size.

Fact 1. For a tree $T$ with $n$ nodes, the total size of all subtrees in $\text{CDecomp}(T)$ is $O(n \log n)$. The decomposition $\text{CDecomp}(T)$ can be computed in $O(n \log n)$ time.
Combinatorics of strings Let $u$ be a string over an integer alphabet $\Sigma$. Then $u = u_1 u_2 \ldots u_n$, where $u_i \in \Sigma$ and $n = |u|$. A substring $u_i \ldots u_j$ of $u$ is called a prefix if $i = 1$ and a suffix if $j = n$. A border of a string $u$ is a string that is both a prefix and a suffix of $u$. We say that $u$ has a period $p$ if $u_i = u_{i+p}$ for all $i = 1, \ldots, n-p+1$. By $u^R$ we denote the string $u_n u_{n-1} \ldots u_1$.

For $u = u_1 u_2 \ldots u_n$, define $rot(u) = u_2 \ldots u_n u_1$. For an integer $q$, let $rot(u, q)$ denote $rot^q(u)$, i.e., the result of $q$ iterations of the $rot$ operation on the string $u$. If $v = rot(u, q)$ for some non-negative integer $q$ then $u$ and $v$ are called cyclically equivalent, we also say that $v$ is a cyclic rotation of $u$. By $maxRot(u)$ we denote the lexicographically maximal cyclic rotation of $u$, see Fig. 2.

A cyclic interval $I$ modulo $n$ is a subset of $\{0, \ldots, n-1\}$ of the form $[a, b]$ or $[a, n-1] \cup [0, b]$. For a cyclic interval $I$, we denote:

$$Rotations(u, I) = \{rot(u, q) : q \in I\}.$$  

Figure 2: We have here $|sq(T)| = 31$. There are 10 groups of cyclically equivalent squares, their representatives (maximal cyclic rotation of a square half) are: $a$, $a^2$, $a^3$, $ba$, $ba^2$, $ba^3$, $ba^4$, $ba^5$, $ba^6$, ($ba^3)^2$.

**Semiruns and anchored squares** Let $T$ be an undirected tree with edges labeled with the symbols from $\Sigma$. Let $u$ and $v$ be two nodes of $T$. By $path(u, v)$ we denote the sequence of nodes in the simple path connecting $u$ and $v$, and by $val(u, v)$ we denote the string obtained by concatenating the labels of edges on this path. Also let $dist(u, v) = |val(u, v)|$.

**Definition 1.** Let $r$ and $v$ be nodes of $T$. By $semirun(r, v)$ we denote the triple $(x, y, p)$ if:

- $r, v \in path(x, y)$,
- $p = dist(r, v)$,
- $|val(x, y)| \geq 2p$,
- $val(x, y)$ has period $p$,
- $path(x, y) = A \cdot B \cdot C$ where $B = path(r, v)$ and $|A|, |C| \leq p$.

If several such triples exist, we select the one with the maximum $|val(x, y)|$. If no such triple exists, we set $semirun(r, v) = \text{nil}$. See also Fig. 3.

Figure 3: The structure of a semirun.
Observation 1. If \((x, y, p)\) is a semirun then all substrings of \(\text{val}(x, y)\) and \(\text{val}(y, x)\) of length \(2p\) are squares. We say that these squares are induced by the semirun.

Let \(v\) be a node of \(T\). A square in \(T\) is called anchored in \(v\) if it is the value of a path passing through \(v\). By \(\text{sq}(T, v)\) we denote the set of squares anchored in \(v\).

Let \(r\) and \(v\) be a pair of different nodes and let \(p = \text{dist}(r, v)\). By \(\text{sq}(T, r, v)\) we denote the set of squares of length \(2p\) that have an occurrence passing through both \(r\) and \(v\).

Observation 2. \(\text{sq}(T, r) = \bigcup_{v \neq r} \text{sq}(T, r, v)\).

**Proof.** Obviously \(\bigcup_{v \neq r} \text{sq}(T, r, v) \subseteq \text{sq}(T, r)\). It suffices to show the opposite inclusion.

Let \(\text{val}(x, y) \in \text{sq}(T, r)\), such that \(\text{path}(x, y)\) passes through \(r\). Let \(p = \text{dist}(x, y)/2\). Note that the path \(\text{path}(x, y)\) contains a node \(v_0\) with \(\text{dist}(r, v_0) = p\). Hence, \(\text{val}(x, y) \in \text{sq}(T, r, v_0)\).

Furthermore, we have the following obvious observation.

Observation 3. The set of squares induced by semirun \((r, v)\) is \(\text{sq}(T, r, v)\).

For a set of semiruns \(S\), let \(\text{sq}(S)\) denote the set of squares induced by at least one semirun in \(S\) and let \(\text{Semiruns}(T, r) = \{\text{semirun}(r, v) : v \neq r\}\). The following lemma states that all semiruns passing through a node represent all square substrings anchored in this node.

**Lemma 1.** Let \(T\) be a tree with a node \(r\). Then \(\text{sq}((\text{Semiruns}(T, r))) = \text{sq}(T, r)\).

**Proof.** It is a consequence of Observations 2 and 3.

**Packages and the set of all squares.** We have seen (Lemma 1) that semiruns can be regarded as a way to represent sets of squares. Nevertheless, this representation cannot be directly used to count the number of different squares and needs to be translated to a cyclic representation (packages). Let \(T\) be a labeled tree and let \(x, y\) be nodes of \(T\) such that \(\text{val}(x, y) = \text{maxRot}(\text{val}(x, y))\). Moreover let \(I\) be a cyclic interval of integers modulo \(\text{dist}(x, y)\). We define a package as a set of cyclically equivalent squares:

\[
\text{package}(x, y, I) = \text{Rotations}(\text{val}(x, y)^2, I).
\]

A family of packages which altogether represent the set of square substrings of \(T\) is called a cyclic representation of squares in \(T\). Such a family is called disjoint if the packages represent pairwise disjoint sets of squares, see Fig. 4.

**Lemma 2.** Let \(T\) be tree and \(r\) be one of its nodes. Let \(S = \text{Semiruns}(T, r)\). There exists a cyclic representation of \(\text{sq}(T, r)\) that contains at most \(2 \cdot |S|\) packages.
Proof. Let \((x, y, p) \in S\). Then \(r \in path(x, y)\) and \(dist(x, y) \geq 2p\), consequently there exists a node \(z\) on \(path(x, y)\) such that \(dist(r, z) = p\) and squares induced by \((x, y, p)\) are all cyclic rotations of \(\alpha^2\) and \(\beta^2\), where \(\alpha = val(r, z)\), \(\beta = val(z, r)\). All cyclic rotations of \(\alpha\) and \(\beta\) occur as substrings of \(path(x, y)\). Let \(x_1, y_1, x_2, y_2 \in path(x, y)\) be nodes such that
\[val(x_1, y_1) = \text{maxRot}(\alpha)\quad\text{and}\quad val(x_2, y_2) = \text{maxRot}(\beta).
\]

Claim 1. Let \(u\) be a string of length \(n\) with period \(p\), \(n \geq 2p\). Let \(v = u_i \ldots u_{i+p-1}\) for some \(i \in \{1, \ldots, p\}\). Then the set of all squares of length \(2p\) that are substrings of \(u\) is
\[\text{Rotations}(v^2, [p-1-i, p-1-i+(n-2p)]).
\]

Using the claim we obtain the cyclic intervals \(I_1\) and \(I_2\) that represent the set of squares induced by \((x, y, p)\) as
\[\text{package}(x_1, y_1, I_1) \cup \text{package}(x_2, y_2, I_2).
\]

As a consequence of Lemmas 1 and 2 and Fact 1 (centroid decomposition) we obtain the existence of a small disjoint cyclic representation of the set of all square substrings in a tree. In the remaining part of the paper we provide a number of algorithmic tools that can be used to efficiently compute this representation.

Theorem 3. Let \(T\) be a labeled tree with \(n\) nodes. There exists a disjoint cyclic representation of all squares in \(T\) of \(O(n \log n)\) size.

Proof. Note that \(sq(T) = \bigcup\{sq(T, r) : (T, r) \in CDecomp(T)\}\). The total size of trees in \(CDecomp(T)\) is \(O(n \log n)\) and for each of them the squares anchored in its root have a linear-size cyclic representation. This gives a cyclic representation of all squares in \(T\) that consists of \(O(n \log n)\) packages.

To obtain a disjoint cyclic representation, we identify packages that correspond to squares of the same substring, compute a union of the cyclic intervals in every set of such packages and divide each such union into a minimal collection of cyclic intervals. The size of the resulting disjoint representation does not exceed the size of the original representation. \(\square\)

3 Algorithmic toolbox for trees

Navigation in trees We recall two widely known tools for rooted trees: the LCA queries and the LA queries. The LCA query given two nodes \(x, y\) returns their lower common ancestor \(LCA(x, y)\). The LA query given a node \(x\) and an integer \(h \geq 0\) returns the ancestor of \(x\) at level \(h\), i.e. with distance \(h\) from the root. After \(O(n)\) preprocessing both types of queries can be answered in \(O(1)\) time \([14, 15]\). These queries enable us to efficiently navigate also in unrooted trees.

Fact 2. Let \(T\) be an unrooted tree with \(n\) nodes. After \(O(n)\) time preprocessing one can answer the following queries in constant time:

(a) for any two nodes \(x, y\) compute \(dist(x, y)\),

(b) for any two nodes \(x, y\) and integer \(0 \leq d \leq dist(x, y)\) compute \(\text{jump}(x, y, d) = \text{the node } z \in path(x, y)\) with \(dist(x, z) = d\).

Proof. Let \(r\) be an arbitrary node of \(T\). We answer queries using an auxiliary tree \(T_r\): a copy of \(T\) rooted in \(r\). For each node \(v \in T_r\) we store its depth, that is, \(dist(r, v)\). We construct the LCA and LA data structures for \(T_r\).

Let \(x, y\) be the query nodes. Let us find the node \(v = LCA(x, y)\). This suffices to answer a \(dist\) query, since
\[dist(x, y) = dist(r, x) + dist(r, y) - 2 \cdot dist(r, v).
\]

To answer a \(jump\) query, we perform a single LA query from \(x\) or from \(y\) depending on whether \(d \leq dist(x, v)\). \(\square\)
**Dictionary of basic factors**  The dictionary of basic factors (DBF, in short) is a widely known data structure for comparing substrings of a string. For a string $w$ of length $n$ it takes $O(n \log n)$ time and space to construct and enables lexicographical comparison of any two substrings of $w$ in $O(1)$ time, see [16]. The DBF can be extended to arbitrary labeled trees.

**Fact 3.** Let $T$ be a labeled tree with $n$ nodes. After $O(n \log n)$ time preprocessing any two substrings $val(x_1, y_1)$ and $val(x_2, y_2)$ of $T$ of the same length can be compared lexicographically in $O(1)$ time (given $x_1, y_1, x_2, y_2$).

*Proof.* Let $T_r$ be a directed labeled tree obtained from $T$ by selecting an arbitrary node $r$ as the root and directing all edges towards the root. For each power of two $2^i$ and node $v \in T_r$, we consider the path of length $2^i$ starting at $v$ and the reversal of the path (if they exist) and assign DBF identifiers $id(v, i)$ and $id^R(v, i)$ to the substrings of $T$ that correspond to such paths. Such identifiers are integers in the range $1, \ldots, 2n$ that preserve the result of lexicographical comparison of substrings of the same length $2^i$. All identifiers are assigned exactly as in the regular DBF in $O(n \log n)$ time, that is, from the shortest to the longest substrings.

![Figure 5: Comparing $val(x_1, y_1)$ and $val(x_2, y_2)$ as in Fact 3. Basic factors that cover the respective parts of the paths are depicted with arrows.](image)

Let $z_1 = LCA(x_1, y_1)$, $z_2 = LCA(x_2, y_2)$, and assume without loss of generality that $dist(x_1, z_1) \geq dist(x_2, z_2)$. Also denote:

$$w_1 = jump(x_1, z_1, dist(x_2, z_2)) \text{ and } w_2 = jump(y_2, z_2, dist(y_1, z_1)).$$

Each of the substrings $val(x_1, w_1)$, $val(w_1, z_1)$ and $val(z_1, y_1)$ can be covered by two basic factors, similarly for the substrings $val(x_2, z_2)$, $val(z_2, w_2)$ and $val(w_2, y_2)$, see also Fig. 5. For example, $val(x_1, w_1)$, with $d = dist(x_1, w_1)$, corresponds to the basic factors:

$$id(x_1, i), \ id(jump(x_1, w_1, d - 2^i), i) \text{ for } 2^i \leq d < 2^{i+1}.$$  

Thus lexicographical comparison of $val(x_1, y_1)$ and $val(x_2, y_2)$ reduces to a comparison of two $6$-tuples of DBF identifiers that can be performed in $O(1)$ time. \hfill $\square$

### 4  The structure of the main algorithm

The main point is efficient computation of the set of semiruns, we postpone its description. The following fact is shown in the following section.

6
Lemma 4. The set Semiruns\((T, r)\) can be computed in \(O(n \log n)\) time.

Let \(T_r\) be a tree rooted at \(r\). We write \(val(v)\) instead of \(val(r, v)\), \(val^R(v)\) instead of \(val(v, r)\) and \(dist(v)\) instead of \(dist(r, v)\).

To translate a representation of squares by semiruns to a representation involving packages we use the following two tables defined for any node \(v\) of \(T_r\).

1. **[Shift Table]**
   
   \(\text{SHIFT}[v]\) is an integer \(q\) such that \(\text{rot}(val(v), q) = \text{maxRot}(val(v))\).

2. **[Reversed Shift Table]** \(\text{SHIFT}^R[v]\) is an integer \(q\) such that

   \[
   \text{rot}(val^R(v), q) = \text{maxRot}(val^R(v)).
   \]

In Section 8 we prove:

**Lemma 5.** The tables \(\text{SHIFT}\), \(\text{SHIFT}^R\) can be computed in \(O(n \log n)\) time.

Using these tables and the \(\text{jump}\) queries (Fact 2) we compute the cyclic representation of the set of squares induced by a family of semiruns.

**Lemma 6.** Let \(T\) be tree and \(r\) be one of its nodes. Let \(S = \text{Semiruns}(T, r)\). A cyclic representation of \(sq(T, r)\) that contains at most \(2 \cdot |S|\) packages can be computed in \(O(n \log n)\) time.

*Proof.* The transformation is performed basically as in the proof of the corresponding combinatorial Lemma 2.

We consider any \((x, y, p) \in S\) and find a node \(z\) on \(\text{path}(x, y)\) such that \(\text{dist}(z) = p\). For this, we need a single \(\text{jump}\) query from either \(x\) or \(y\). Next we use the \(\text{SHIFT}\) and \(\text{SHIFT}^R\) tables to locate the occurrences of \(\text{maxRot}(\alpha) = \text{maxRot}(\text{val}(z))\) and \(\text{maxRot}(\beta) = \text{maxRot}(\text{val}^R(z))\). Then \(\text{jump}\) queries allow to find the exact endpoints \(x_1, y_1\) and \(x_2, y_2\) of the occurrences of these maximal rotations. The cyclic intervals \(I_1, I_2\) for the cyclic representation \(\text{package}(x_1, y_1, I_1) \cup \text{package}(x_2, y_2, I_2)\) are computed as described in the claim in Lemma 2. \(\square\)

The general structure of the main algorithm is based on centroid decomposition.

**Algorithm 1: Count-Squares(T)**

```
foreach \((T, r) \in \text{CDecomp}(T)\) do
    \text{Semiruns} := \text{Semiruns}(T, r) \{} \text{using Lemma 4}\{} \\
    \text{Transform} \text{Semiruns} \text{ into a set of packages in } T \text{ (Lemma 6)} \\
    \text{Insert these packages to the set } Packages

\text{Compute disjoint representation of } Packages
\text{return } |sq(T)| \text{ as the total length of intervals in } Packages
```

The complexity of the \texttt{Count-Squares(T)} algorithm is analyzed in the following main theorem.

**Theorem 7.** The number of distinct square substrings in an (unrooted) tree with \(n\) nodes together with a disjoint cyclic representation of these squares of size \(O(n \log n)\) can be found in \(O(n \log^2 n)\) time.

*Proof.* The total size of trees in the centroid decomposition of \(T\) is \(O(n \log n)\). Now we consider each pair \((T, r) \in \text{CDecomp}(T)\) separately. Let \(m\) be the number of nodes of \(T\). We need to show that the computations for \((T, r)\) can be performed in \(O(m \log m)\) time.

First we build the rooted tree \(T_r\) and for this tree build the data structures for DBF, \(\text{dist}\) and \(\text{jump}\) queries and the \(\text{SHIFT}\) and \(\text{SHIFT}^R\) tables By Fact 2, Fact 3 and Lemma 5 respectively this requires \(O(m \log m)\) time in total.
Next we apply Lemma 4 to compute $\text{Semiruns}(T, r)$ in $O(m \log m)$ time. By Lemma 6, the semiruns can be transformed to a cyclic representation of squares in $T$ of size $O(m)$. This, again, requires $O(m \log m)$ time.

Finally we compute a disjoint cyclic representation of all squares in $T$ (that is, across all pairs $(T, r)$). For this, we group packages $(x, y, I)$ in the cyclic representation according to $\text{val}(x, y)$, which is done by sorting them using Fact 3 for the comparison criterion. Finally in each group by elementary computations we turn a union of arbitrary cyclic intervals into a union of pairwise disjoint intervals. For a group of $g$ packages this is done in $O(g \log g)$ time, which makes $O(n \log^2 n)$ in total. \qed

The complete proof of Theorem 7 requires only justification of two lemmas: Lemma 4 and Lemma 5. The following sections are doing this job.

5 Computing semiruns

5.1 Determinization of a tree

Let $T_r$ be a tree rooted in $r$. The tree $T_r$ is said to be deterministic if $\text{val}(v) = \text{val}(w)$ implies that $v = w$. $T_r$ is semideterministic if $\text{val}(v) = \text{val}(w)$ implies that $v = w$ or $\text{path}(r, v)$ and $\text{path}(r, w)$ are disjoint except $r$. Hence, $T_r$ is semideterministic if it is “deterministic anywhere except for the root”.

For an arbitrary tree $T_r$, an “equivalent” deterministic tree $\text{Deter}(T_r)$ can be obtained by identifying nodes $v, w$ if $\text{val}(v) = \text{val}(w)$. If we perform such identification only when the paths $\text{path}(r, v)$ and $\text{path}(r, w)$ share the first edge, we obtain a semideterministic tree $\text{SemiDeter}(T_r)$. This way we also obtain functions $\varphi_d (\varphi_s$ respectively) mapping nodes of $T_r$ to corresponding nodes in $\text{Deter}(T_r)$ (in $\text{SemiDeter}(T_r)$ respectively). Additionally we define $\psi_d(v)$ ($\psi_s(v)$ respectively) as an arbitrary element of $\varphi_d^{-1}(v)$ for $v \in \text{Deter}(T_r)$ ($\varphi_s^{-1}(v)$ for $v \in \text{SemiDeter}(T_r)$ respectively).

![Figure 6: Different types of tree determinization.](image)

**Fact 4.** Let $T_r$ be a rooted tree of $n$ nodes. Then $\text{Deter}(T_r)$ and $\text{SemiDeter}(T_r)$ together with the corresponding pairs of functions $\varphi_d, \psi_d$ and $\varphi_s, \psi_s$ can be computed in $O(n)$ time.

**Proof.** We first show how to compute the determined tree $\text{Deter}(T_r)$. For each node $v$ of $T_r$ we compute the node $\varphi_d[v]$ of $\text{Deter}(T_r)$, corresponding to $v$ in the determined tree. We also compute an auxiliary table $\text{children}[v]$ for each node $v$ in $\text{Deter}(T_r)$ containing the list of edges going down from $v$ in $\text{Deter}(T_r)$, sorted by their labels.
Algorithm 2: Compute $Deter(T_r)$ for $T_r$

| sort all edges in $T_r$ by label and store them in $E$ |
| stably sort $E$ by the depth of the edges (edges closest to $r$ come first) |
| initialize $children$ to be empty |
| $\varphi_d[r] := r$ |
| \hspace{1em} foreach $(c, u, v) \in E$ do |
| \hspace{2em} if $(c, w) = \text{last_element}(\text{children}[\varphi_d[u]])$ for some $w$ then |
| \hspace{3em} \{ if $(c, w)$ is in the list, it must be the last element $\}$ |
| \hspace{3em} $\varphi_d[v] := w$ |
| \hspace{2em} else |
| \hspace{3em} $\varphi_d[v] := v$ |
| \hspace{3em} append$(\text{children}[\varphi_d[u]], (c, \varphi_d[v]))$ |

Counting sort can be employed for sorting the edges; consequently, Algorithm 2 works in linear time.

To compute $SemiDeter(T_r)$ it suffices to apply Algorithm 2 to all subtrees rooted at children of $r$. \hfill $\square$

**Observation 4.** Note that $\varphi_s$ and $\psi_s$ preserve the values of paths going through $r$; this property does not hold for $\varphi_d$ and $\psi_d$, since children of $r$ may get glued together.

### 5.2 Two basic tables

Consider a rooted tree $T_r$. In order to prove Lemma 4 we introduce two tables, defined for all $v \neq r$, similar to the tables used in Main-Lorenz square-reporting algorithm for strings [17].

- **[Prefix table]** $PREF[v]$ is a lowest node $x$ in the subtree rooted at $v$ such that $val(v, x)$ is a prefix of $val(v)$, see Fig. 7a.

- **[Suffix table]** $SUF[v]$ is a lowest node $x$ in $T_r$ such that $val(x)$ is a prefix of $val^R(v)$ and $LCA(v, x) = r$, see Fig. 7b.

In Sections 6 and 7 we prove the following lemma.

**Lemma 8.** For a semideterministic rooted tree $T_r$, the $PREF$ and $SUF$ tables can be computed in linear time.

### 5.3 The Proof of Lemma 4

Let us note that the $PREF$ and $SUF$ tables provide a characterization of semiruns (see also Fig. 8).
Observation 5. Consider $v \neq r$. Then

$$\text{semirun}(r,v) = (\text{SUF}[v], \text{PREF}[v], \text{dist}(v))$$

provided that the semirun exists.

of Lemma 4. Let $T$ be a tree an $r$ be one of its nodes. Let $T'_r$ be a copy of $T$ rooted in $r$ and let $T'_r = \text{SemiDeter}(T_r)$. By Lemma 8, one can compute the PREF and SUF tables for $T'_r$ in linear time. The following algorithm uses Observation 5 to compute $\text{Semiruns}(T'_r, r)$ and returns the set:

$$S = \{(\psi_s(x), \psi_s(y), p) : (x,y,p) \in \text{Semiruns}(T'_r, r)\}.$$  

Due to Observation 4, we have $S = \text{Semiruns}(T, r)$.

Algorithm 3: Compute $\text{Semiruns}(T, r)$

$$S := \emptyset; \quad T'_r := \text{SemiDeter}(T_r)$$

Compute the tables $\text{PREF}, \text{SUF}$ for $T'_r$

$$\text{foreach } v \in T'_r \setminus \{r\} \text{ do}$$

$$x := \text{PREF}[v]; \quad y := \text{SUF}[v]$$

$$\text{if } \text{dist}(x,y) \geq 2 \cdot \text{dist}(v) \text{ then}$$

$$S := S \cup \{(\psi_s(y), \psi_s(x), \text{dist}(v))\}$$

$$\text{return } S$$

This completes the proof of Lemma 4 provided that we have the $\text{PREF}$ and $\text{SUF}$ tables.

6 Computation of $\text{PREF}$

The $\text{PREF}$ and $\text{SUF}$ tables for ordinary strings are computed by a single simple algorithm, see [16]. This approach fails to generalize for trees, so we develop novel methods, interestingly, totally different for both tables. In order to construct a $\text{PREF}$ table we generalize the results of Simon [18] originally developed for string pattern matching automata. For the $\text{SUF}$ table, we use the suffix tree of a tree, a concept originally designed for tree pattern matching that was introduced by Kosaraju [19] and later on an improved construction time was given by Shibuya [20].

Let $T_r$ be a rooted semideterministic tree. We compute a slightly modified array $\text{PREF}'$ that allows for an overlap of the considered paths. More formally, for a node $v \neq r$, we define $\text{PREF}'[v]$ as the lowest node $x$ in the subtree rooted at $v$ such that $\text{val}(v,x)$ is a prefix of $\text{val}(x)$. Note that having computed $\text{PREF}'$, we can obtain $\text{PREF}$ by truncating the result so that the paths do not overlap. This can be implemented with a single $\text{jump}$ query.
Note that $PREF^t[v]$ depends only on the path $path(r, v)$ and the subtree rooted at $v$. Hence, instead of a single semideterministic tree of $n$ nodes, we may create a copy of $r$ for each edge going out from $r$ and thus obtain several deterministic trees of total size $O(n)$. For the remainder of this section we assume $T_r$ is deterministic.

The $PREF$ function for strings is closely related to borders, see [16]. This is inherited by $PREF'$ for deterministic trees which we state as the following Fact 5 (see also Fig. 9). Let next($x$) denote the set of labels of edges leaving $x$.

![Figure 9: $PREF'[v] = x$ if and only if $val(v, x)c$ is a border of $val(x)c$ and no edge labeled with $c$ leaves $x$.](image)

**Fact 5.** Let $T_r$ be a deterministic tree rooted at $r$. Let $v \neq r$ be a node in $T_r$ and let $x$ be its descendant. Then $PREF'[v] = x$ if and only if $val(v, x)c$ is a border of $val(x)c$ for some $c \in \Sigma \setminus \text{next}(x)$.

*Proof.* Note that in a deterministic tree $PREF'[v]$ can be (inefficiently) computed by the following procedure. Start with $x := r$ and $y := v$. Let $a$ be the first letter of $val(x, y)$. If $a \in \text{next}(x)$, move $x$ and $y$ one level down following the $a$-labeled edges and repeat the procedure. Otherwise we set $PREF'[v] := x$. In a deterministic tree each step of this procedure is uniquely determined, which easily implies the correctness of this procedure. Now the statement of the fact is equivalent to a halting condition of the procedure. □

![Figure 10: We have: $\pi[4] = \{(a, 2), (b, 3)\}$, $\pi[7] = \{(a, 5), (b, 3)\}$, $P[7] = 4$, $PREF'[3] = 4$, $PREF'[4] = 7$.](image)

For a node $x$ of $T_r$ and $c \in \Sigma$, let $\pi(x, c)$ (transition function) be a node $y$ such that $val(y)$ is the longest border of $val(x)c$. We say that $\pi(x, c)$ is an essential transition if it does not point to the root. Let us define the transition table $\pi$ and the border table $P$. For a node $x$ let $\pi[x]$ be the list of pairs $(c, y)$ such that $\pi(x, c) = y$ is an essential transition. For $x \neq r$ we set $P[x]$ as the node $y$ such that $val(y)$ is the longest proper border of $val(x)$. The following fact generalizes the results of [18] and gives the crucial properties of essential transitions.

**Fact 6.** Let $T_r$ be a deterministic rooted tree with $n$ nodes. There are no more than $2n - 1$ essential transitions in $T_r$.

*Proof.* Clearly the number of essential transitions $\pi(x, c)$ such that $c \in \text{next}(x)$ is bounded by $n - 1$, the number of edges. Let us construct a one-to-one function $F$ mapping the remaining essential transitions to the nodes of $T$. Let $\pi(x, c) = y$ be an essential transition such that $c \notin \text{next}(x)$. Let $v$ be the only ancestor of $x$ such that $\text{dist}(v, x) = \text{dist}(y) - 1$. This is precisely the situation from Fact 5, so $PREF'[v] = x$. We set $F(x, c) = v$. Note that, in deterministic trees, $x = PREF'[v]$ is uniquely determined by $v$ and, moreover, $c$ is the only letter such that $val(v, x)c$ is a border of $val(x)c$. Hence $F$ is indeed one-to-one and there are at most $n$ essential transitions $\pi(x, c)$ with $c \notin \text{next}(x)$. This concludes the proof of the fact. □
Before we present an algorithm computing $\pi$ and $P$ tables, let us introduce additional notation. We say that $L$ is a dictionary list if $L$ is a sorted list of pairs $(c, w)$ with unique $c$. If $(c, w)$ is a member of $L$, we say that $L$ maps $c$ into $w$.

For a node $x$ of $T$, let $\text{children}[x]$ be a dictionary list mapping $c \in \text{next}(x)$ into the corresponding child nodes of $x$. Note that our determinization algorithm actually computes such lists.

**Lemma 9.** Let $T_r$ be a deterministic rooted tree with $n$ nodes. The $\pi$ and $P$ tables for $T_r$ can be computed in $O(n)$ time.

**Proof.** In the algorithm we extensively use the dictionary lists $\text{children}[x]$. The transition lists $\pi$ that we compute are also dictionary lists. For two dictionary lists $L_1, L_2$ indexed by $\Sigma$ we define $L = \text{merge}(L_1, L_2)$ as the “outer join” of $L_1$ and $L_2$. More precisely, $L$ maps $c$ into $(w_1, w_2)$ if $L_1$ maps $c$ into $w_1$ and $L_2$ maps $c$ into $w_2$. If one of the lists does not map $c$ into anything, but the other does, we set the corresponding $w_i$ to nil. Note that the time complexity of computing $\text{merge}(L_1, L_2)$ is proportional to the total size of both lists.

**Algorithm 4:** Compute the $\pi$ and $P$ tables for $T_r$

```plaintext
\pi[r] := empty table
foreach $(c, w) \in \text{children}[r]$ do
    $P[w] := r$
foreach $x \in T_r \setminus \{r\}$ (in preorder) do
    $y := P[x]$
    $y' := \text{jump}(y, x, 1)$
    $a := \text{val}(y, y')$
    $\pi[x] := \text{empty table}$
    foreach $(b, u) \in \pi[y]$ do
        if $a \neq b$ then
            $\text{append}(\pi[x], (b, u))$
        insert($\pi[x], (a, y'))$
        foreach $(b, (u, w)) \in \text{merge}(\pi[x], \text{children}[x])$ do
            if $w \neq \text{nil}$ then
                if $u \neq \text{nil}$ then $P[w] := u$
                else $P[w] := r$
```

Algorithm 4 computes the $\pi$ and $P$ tables by definition. Here the order of computations is crucial. When we visit a node $x$, we compute $\pi[x]$ and fill the border table $P$ for all children of $x$. We assume that $\pi$ and $P$ were already computed for proper ancestors of $x$ and $P$ was computed for $x$. Time complexity of such a single step is proportional to the total size of $\pi[x]$ and $\text{children}[x]$, which sums up to $O(n)$ over all nodes $x$.

**Lemma 10.** For a deterministic tree $T_r$, the table $\text{PREF}'$ can be computed in linear time.

**Proof.** Algorithm 5 uses the characterization of $\text{PREF}'$ given by Fact 5. For each $x$ we find all nodes $v$ such that $\text{PREF}'[v] = x$.

The algorithm iterates over all borders of $\text{val}(x)c$ for $c \notin \text{next}(x)$. The longest one is found using the transition table $\pi$. The remaining ones are computed by iterating the border table $P$.  

---

12
Let \( n \) be the number of nodes of \( T_r \). For each node \( v \) of \( T_r \) we perform the assignment \( \text{PREF}'[v] := x \) only once, so the total number of steps of the while-loop is \( O(n) \). The complexity of the remaining part of the algorithm is bounded by the total size of the \( \pi \) and \( \text{children} \) tables, which is also \( O(n) \).

Lemma 10 concludes the “\( \text{PREF} \)” part of Lemma 4.

### 7 Computation of \( \text{SUF} \)

Let \( T'_r \) be a deterministic tree rooted at \( r \) and \( v \neq r \) be a node of \( T'_r \). We define \( \text{SUF}'[v] \) as the lowest node \( x \) of \( T'_r \) such that \( \text{val}(x) \) is a prefix of \( \text{val}^R(v) \). Hence, we relax the condition that \( \text{LCA}(u, x) = r \) and add a requirement that the tree is deterministic.

**Lemma 11.** Let \( T_r \) be an arbitrary rooted tree of \( n \) nodes. The \( \text{SUF} \) table for \( T_r \) can be computed in \( O(n) \) time from the \( \text{SUF}' \) table for \( \text{Deter}(T_r) \).

**Proof.** Recall the \( \varphi_d \) function mapping a node of \( T_r \) to the corresponding node in \( \text{Deter}(T_r) \). For a node \( x \neq r \) of \( T_r \) let \( \text{subroot}(x) \) be the child of \( r \) lying on \( \text{path}(r, x) \). For a node \( y' \neq r \) of \( \text{Deter}(T_r) \) let \( \text{subroots}(y') = \{ \text{subroot}(y) : y \in \varphi_d^{-1}(y') \} \). All the subroots can easily be precomputed in linear time. Moreover, together with \( z \in \text{subroots}(y') \) we can store \( y \in \varphi_d^{-1}(y') \) such that \( \text{subroot}(y) = z \).

Using these functions \( \text{SUF}[x] \) can be defined as the lowest node \( y \) such that \( \text{subroot}(y) \neq \text{subroot}(x) \) and \( \varphi_d(y) \) is an ancestor of \( y' = \text{SUF}'[\varphi_d(x)] \). Note that the \( \text{subroots} \) function is monotonic, so either \( \varphi_d(y) = y' \) or \( \text{subroots}(y') = \{ \text{subroot}(x) \} \) and \( \varphi_d(y) \) is the lowest ancestor of \( y' \) whose \( \text{subroots} \) set contains at least two elements. Such ancestors can be precomputed for all nodes of \( \text{Deter}(T_r) \) by a single top-down tree traversal.

Once we know \( z' = \varphi_d(y) \) we pick any element of \( \text{subroots}(z') \) different from \( \text{subroot}(x) \). If \( \text{subroots} \) is implemented as a linked list, it suffices to inspect up to two first elements. Finally, we set \( \text{SUF}[x] = z \), where \( z \in \varphi_d^{-1}(z') \) is the node associated with this subroot.

Recall that a trie of a set of strings is a minimal deterministic rooted labeled tree containing all these strings as paths starting from the root.

**Observation 6.** Let \( S_1 = \{ \text{val}(x) : x \in T_r \} \) and \( S_2 = \{ \text{val}^R(x) : x \in T_r \} \). Let \( T \) be a trie of all the strings \( S_1 \cup S_2 \). Then for any \( v \in T_r \), \( \text{SUF}'[v] \) corresponds to the lowest ancestor of \( \text{val}^R(v) \) in \( T \) of the form \( \text{val}(x) \).

Assume that we store the pointers to nodes in \( T \) that correspond to elements of \( S_1 \) and \( S_2 \). Then the ancestors mentioned in Observation 6 can be computed by a single top-bottom tree traversal, so the \( \text{SUF}' \) table can be computed in time linear in \( T \).

Unfortunately, the size of \( T \) can be quadratic, so we store its compacted version in which we only have explicit nodes corresponding to \( S_1 \cup S_2 \) and branching nodes (that is, nodes having at least two children). The trie of \( S_1 \) is exactly \( T_r \), whereas the compacted trie of \( S_2 \) is known as a suffix trie of the tree \( T_r \). This notion was introduced in [19] and a linear time construction algorithm for an integer alphabet was given in [20]. The compacted trie \( T \) can therefore be obtained by merging \( T_r \) with its suffix tree, i.e., identifying nodes of the same value. Since \( T_r \) is not compacted, this can easily be done in linear time. This gives a linear
time construction of the compacted $T$ which yields a linear time algorithm constructing the $SUF^T$ table for $T$, and consequently the following result:

**Lemma 12.** The $SUF$ table of a rooted tree can be computed in linear time.

Lemma 12 concludes the proof of Lemma 4. Note that in $SUF$ computation we do not require the tree to be semideterministic.

**8 Computation of shift tables**

In this section we give the proof of Lemma 5. The computation of maximal rotation (shift) of $w$ is equivalent to finding $\text{maxSuf}(ww)$, see [16].

**Definition 2.** A suffix $u$ of the string $w$ is redundant if for every string $z$ there exists another suffix $v$ of $w$ such that $vz > uz$. Otherwise we call $u$ non-redundant.

**Observation 7.** (a) If $u$ is a redundant suffix of $w$, then for any string $z$ it holds that $uz$ is a redundant suffix of $wz$ and $u$ is a redundant suffix of $zw$. (b) If $u$ is a non-redundant suffix of $w$, then $u$ is a prefix, and therefore a border, of $\text{maxSuf}(w)$.

Before we proceed with the proof, let us introduce a notion of square-centers and its relation with redundancy. A position $i$ in a string $w$ is a square-center if there is a square in $w$ such that its second half starts at $i$.

**Fact 7.** If $i$ is the first position of $\text{maxSuf}(w)$ then $i$ is not a square-center.

**Proof.** Let $w = uxxv$, where $|ux| = i - 1$. We need to show that $xxv$ is not a maximum suffix of $w$. This holds because either $v > xv$ or $v < xv$ and consequently $xxv < xxv$ — in both cases we obtain a lexicographically greater suffix. □

**Lemma 13 (Redundancy Lemma).** If $u, v$ are borders of $\text{maxSuf}(w)$ such that $|u| < |v| \leq 2|u|$ then $u$ is a redundant suffix of $w$.

**Proof.** Due to Fine & Wilf’s periodicity lemma [16] such a pair of borders induces a period of $v$ of length $|v| - |u| \leq |u|$. This concludes that there is a square in $w$ centered at the position $|w| - |u| + 1$. Hence, for any string $z$, the starting position of the suffix $uz$ in $wz$ is a square-center, so, by Fact 7, $uz$ is not the maximal suffix of $wz$. □

**Definition 3.** We call a set $\text{Cand}(w)$ a small candidate set for a string $w$ if $\text{Cand}(w)$ is a subset of suffixes of $w$, contains all non-redundant suffixes of $w$ and $|\text{Cand}(w)| \leq \max(1, \log |w| + 1)$.

**Lemma 14.** Assume we are given a string $w$ together with the DBF of $w$. Then for any $a \in \Sigma$, given small candidate sets $\text{Cand}(w)$ and $\text{Cand}(w^R)$ we can compute $\text{Cand}(wa)$ and $\text{Cand}((wa)^R)$ in $O(\log |w|)$ time.

**Proof.** We represent the sets $\text{Cand}$ as sorted lists of lengths of the corresponding suffixes. For $\text{Cand}(wa)$ we apply the following procedure, see Fig. 11.

1. $C := \{wa : v \in \text{Cand}(w)\} \cup \{\varepsilon\}$, where $\varepsilon$ is an empty string.
2. Determine the lexicographically maximal element of $C$, which must be equal to $\text{maxSuf}(wa)$ by definition of redundancy.
3. Remove from $C$ all elements that are not borders of $\text{maxSuf}(wa)$.
4. While there are $u, v \in C$ such that $|u| < |v| \leq 2|u|$, remove $u$ from $C$.
5. $\text{Cand}(wa) := C$. 

14
Figure 11: The computation of Cand sets as in the proof of Lemma 14. (a) Cand set for the word ababaabab. (b) Cand set for the word ababaababa, the suffixes of length 1 and 2 are removed due to point 4 of the procedure. (c) Cand set for the word ababaababab, here maxSuf changes and the suffix of length 10 is removed due to point 3 of the procedure.

All steps can be done in time proportional to the size of C. It follows from Lemma 13 that the resulting set \( \text{Cand}(wa) \) is a small candidate set. \( \text{Cand}((wa)^R) \) is computed in a similar way.

Lemma 14 provides the tool for computation of the shift tables.

**Lemma 5.** Let \( T_r \) be a rooted tree. We traverse the tree \( T_r \) in DFS order of the nodes and compute \( \text{maxSuf}(ww) \) for each prefix path as:

\[
\text{maxSuf}(ww) = \max\{yw : y \in \text{Cand}(w)\}.
\]

Here we use tree DBF and jump queries for lexicographical comparison. If we know \( \text{maxSuf}(ww) \), maximal cyclic shift of \( w \) is computed in \( O(1) \) time.

9 Linear time algorithm for combs

There is an interesting class of trees called combs which are rich in squares — they have \( \Omega(n^{4/3}) \) distinct squares, it has been recently shown in [7] that asymptotically it is also an upper bound for all trees. A comb consists of a single path (called spine) with all edges labeled \( a \) and outgoing branches, labeled by \( b \) on the first edge of the branch and \( a \) on other edges. Despite the large number of squares in combs they can be counted in linear time.

**Theorem 15.** If \( T \) is a comb then \( sq(T) \) can be computed in linear time.

**Proof.** We say that a pair \( u, v \) of nodes on the spine is admissible iff \( d \leq i + j \) where \( d = \text{dist}(u, v) \) and \( i, j \) are numbers of \( a \)'s on the branches attached to \( u \) and \( v \). When finding all admissible pairs we can consider
Figure 12: An essential part of a comb corresponding to nodes \(u, v\) on the spine consists of edges between them and outgoing branches starting with the letter \(b\). The set of non-unary squares generated by this part equals \(\{a^k ba^k a^{d-k} ba^{d-k} : k \in I(u,v)\}\), where \(d = 6\) and \(I(u,v) = [1,3]\).

Each node \(u\) on the spine and assume that the length of the branch at \(v\) is at most that at \(u\). Hence, for each node \(u\) on the spine it is enough to process nodes \(v\) on the spine at distance at most \(2i\) from \(u\), where \(i\) is the number of \(a\)'s on the branch outgoing from \(u\). Such numbers are amortized by the lengths of outgoing branches, the sum of these lengths is linear. Consequently we have the following fact:

**Claim 2.** The number of admissible pairs is linear and all of them can be computed in linear time.

We can group admissible pairs into sets with the same distance \(d\) between the nodes in the pair. For each pair \((u,v)\) the set of squares generated by this pair, see Fig. 12, corresponds to an interval. These intervals (for distinct pairs) are not necessarily disjoint, however we can make a union of all intervals in linear time, the resulting set is again a union of intervals and its total size can be easily computed. The sets of squares corresponding to distinct groups are disjoint. We sum the numbers for each group and get the final result. This completes the proof. \(\Box\)

**Observation.** Despite the fact that we can have superlinearly many distinct squares all of them can be reported as a union of linearly many disjoint sets of the form \(\{a^k ba^k a^{d-k} ba^{d-k} : k \in [l,r]\}\).

**References**


