Efficient Counting of Square Substrings in a Tree☆

Tomasz Kociumaka[a,1], Jakub Pachocki[b,2], Jakub Radoszewski[a,3,*], Wojciech Rytter[a,c], Tomasz Waleń[a]

[a] *Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Poland*
[b] *Carnegie Mellon University*
[c] *Faculty of Mathematics and Computer Science, Copernicus University, Toruń, 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.

Keywords: tree, square in string, pattern matching

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].

☆A preliminary version of this paper appeared at ISAAC 2012 [1].

*Corresponding author

Email addresses: `kociumaka@mimuw.edu.pl` (Tomasz Kociumaka), `pachocki@cs.cmu.edu` (Jakub Pachocki), `jrad@mimuw.edu.pl` (Jakub Radoszewski), `rytter@mimuw.edu.pl` (Wojciech Rytter), `walen@mimuw.edu.pl` (Tomasz Waleń)

[1] Supported by Polish budget funds for science in 2013-2017 as a research project under the 'Diamond Grant' program.

[2] This work was done while at University of Warsaw.

[3] The author receives financial support of Foundation for Polish Science.

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 Σ. We assume that Σ is polynomially bounded in terms of n, i.e. $\Sigma \subseteq \{0, \ldots, n^C\}$ 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 $\mathrm{sq}(T)$ the set of different square substrings in T. The main problem we consider is as follows:

Input: A labeled tree T.

Output: $|\mathrm{sq}(T)|$, the number of distinct square substrings in T.

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

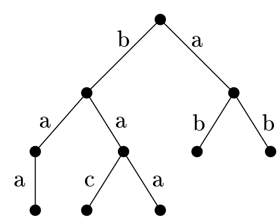


Figure 1: This tree contains the following squares: *aa*, *aaaa*, *abab*, *baba*, *bb*. We have here $|\mathrm{sq}(T)| = 5$. Note that the squares *abab* and *baba* correspond to the same path, one is read from left to right and the other is read from right to left.

Our result: We compute $|\mathrm{sq}(T)|$ 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 $u = val(x, y)$ 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 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 substrings [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 $|T_i| \le n/2$ for all T_i. The *centroid decomposition* of T, $\mathit{CDecomp}(T)$, is defined recursively:

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

Note that for every path p in T there exists an element $(T', r') \in \mathit{CDecomp}(T)$ such that p is a path in T' that passes through r'. This can be proved by a simple induction on $|T|$: either p passes through r in T, or we use the inductive hypothesis for the subtree T_i that contains p.

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

Fact 1. *For a tree T with n nodes, the total size of all subtrees in $\mathit{CDecomp}(T)$ is $O(n \log n)$. The decomposition $\mathit{CDecomp}(T)$ can be computed in $O(n \log n)$ time.*

Combinatorics of strings. Let u be a string over an integer alphabet Σ. 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 $\mathit{rot}(u) = u_2 \ldots u_n u_1$. For an integer q, let $\mathit{rot}(u, q)$ denote $\mathit{rot}^q(u)$, 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 $[a,b]$ of $\{0,\dots,n-1\}$ of the form $\{a,\dots,b\}$ (if $b \geq a$) or $\{a,\dots,n-1,0,\dots,b\}$ (if $b < a$). For a cyclic interval I, we denote:

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

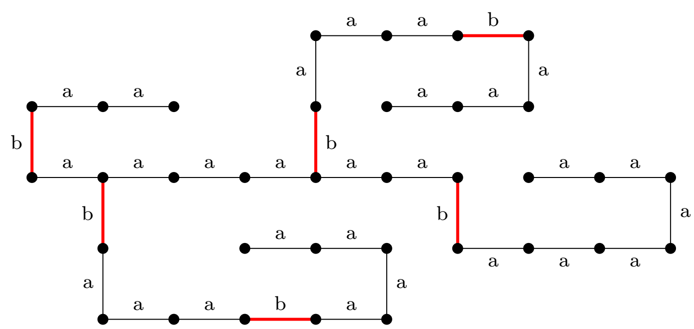


Figure 2: We have here $|\mathrm{sq}(T)| = 31$. There are 10 groups of cyclically equivalent squares. The maximal cyclic rotations of their halves are: a, a^2, a^3, ba, ba^2, ba^3, ba^4, ba^5, ba^6, $(ba^3)^2$; e.g. $(aabaa)^2$ is the only square substring of T whose half is cyclically equivalent to ba^4.

Semiruns and anchored squares. Let T be an undirected tree with edges labeled with the symbols from Σ. 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:*

$$r, v \in path(x,y), \quad p = dist(r,v), \quad dist(x,y) \geq 2p,$$

$$dist(x,r) \leq p, \quad dist(v,y) \leq p, \quad val(x,y) \text{ has period } p.$$

*If several such triples exist, we select the one with the maximum $dist(x,y)$. If no such triple exists, we set $semirun(r,v) =$ **nil**. See also Fig. 3.*

Observation 1. *If (x,y,p) is a semirun then all substrings of $val(x,y)$ and $val(y,x)$ 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 $\mathrm{sq}(T,v)$ we denote the set of squares anchored in v.

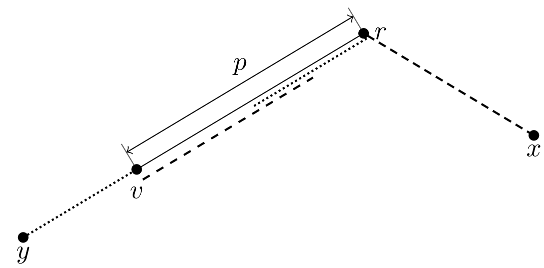


Figure 3: The structure of a semirun. Dashed and dotted segments represent paths labeled with equal strings.

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

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

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

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

Furthermore, we have the following obvious observation.

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

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

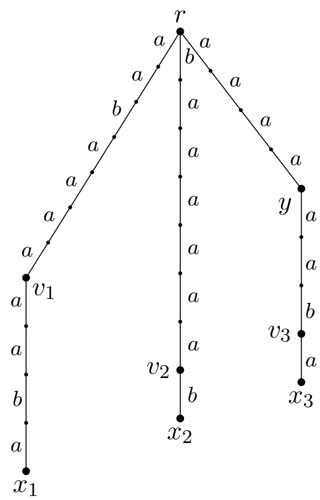
Lemma 1. *Let T be a tree with a node r. Then* $\mathrm{sq}(Semiruns(T, r)) = \mathrm{sq}(T, r)$*.*

Proof. It is a consequence of Observations 2 and 3. $\square$

Packages and the set of all squares. We have seen in 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 converted to a cyclic representation (packages). Let T be a labeled tree and let x, y be nodes of T such that $val(x, y) = maxRot(val(x, y))$. Moreover let I be a cyclic interval of integers modulo $dist(x, y)$. We define a *package* as a set of cyclically equivalent squares:

$$package(x, y, I) = Rotations(val(x, y)^2, I).$$

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.



Semiruns and packages induced by them:

$semirun(r, v_1) = (y, x_1, 7) \quad \rightsquigarrow$

$package(baaaaaa, [1, 2]),\ package(baaaaaa, [6, 0])$

$semirun(r, v_2) = (v_3, x_2, 7) \quad \rightsquigarrow$

$package(baaaaaa, [0, 1]),\ package(baaaaaa, [0, 1])$

$semirun(r, v_3) = (v_2, x_3, 7) \quad \rightsquigarrow$

$package(baaaaaa, [1, 2\}),\ package(baaaaaa, [6, 0])$

Disjoint cyclic representation:

$package(baaaaaa, [6, 2])$

Figure 4: This tree contains 4 squares that are cyclic rotations of $(baaaaaa)^2$.

Lemma 2. *Let T be tree and r be one of its nodes. Let $S = Semiruns(T, r)$. There exists a cyclic representation of $\mathrm{sq}(T, r)$ 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 α^2 and β^2, where $\alpha = val(r, z)$, $\beta = val(z, r)$. All cyclic rotations of α and β occur as substrings of $path(x, y)$. Let $x_1, y_1, x_2, y_2 \in path(x, y)$ be nodes such that

$$val(x_1, y_1) = maxRot(\alpha) \quad \text{and} \quad val(x_2, y_2) = maxRot(\beta).$$

Claim 1. *Let u be a string of length n with period p, $n \geq 2p$. Let $v = u_i \dots u_{i+p-1}$ for some $i \in \{1, \dots, p\}$. Then the set of all squares of length $2p$ that are substrings of u is*

$$Rotations(v^2,\ [p + 1 - i,\, p + 1 - i + (n - 2p)]).$$

Proof. Due to the periodicity of u, we have $v = u_i \dots u_p u_1 \dots u_{i-1}$. Note that u starts with $(u_1 \dots u_p)^2 = rot(v, p + 1 - i)^2$. In total u contains $n - 2p + 1$ substrings of length p, that are consecutive cyclic rotations of v^2. This yields the interval of cyclic rotations as stated in the claim. □

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

$$package(x_1, y_1, I_1) \cup package(x_2, y_2, I_2).$$

□

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 $\boldsymbol{T}$ be a labeled tree with n nodes. There exists a disjoint cyclic representation of all squares in $\boldsymbol{T}$ of $O(n \log n)$ size.*

PROOF. Note that $\mathrm{sq}(\mathbf{T}) = \bigcup\{\mathrm{sq}(T, r) : (T, r) \in \mathit{CDecomp}(\mathbf{T})\}$. The total size of trees in $\mathit{CDecomp}(\mathbf{T})$ 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 $\mathbf{T}$ 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 $jump(x, y, d)$ — 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.

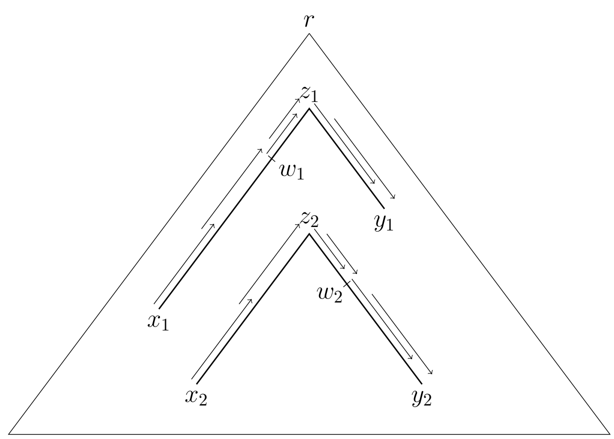


Figure 5: Comparing $val(x_1, y_1)$ and $val(x_2, y_2)$ a in Fact 3. Basic factors that cover the respective parts of the paths are depicted with arrows.

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:

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

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. □

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.

Lemma 4. *The set $Semiruns(T, r)$ can be computed in $O(n)$ 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 convert a representation of squares by semiruns to the representation involving packages we use the following two tables defined for any node v of T_r.

1. **[Shift Table]**
   $SHIFT[v]$ is an integer q such that $rot(val(v), q) = maxRot(val(v))$.
2. **[Reversed Shift Table]** $SHIFT^R[v]$ is an integer q such that
$$rot(val^R(v), q) = maxRot(val^R(v)).$$

In Section 8 we prove:

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

Using these tables and the *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 $S = Semiruns(T, r)$. A cyclic representation of $\mathrm{sq}(T, r)$ 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 $path(x, y)$ such that $dist(z) = p$. For this, we need a single *jump* query from either x or y. Next we use the $SHIFT$ and $SHIFT^R$ tables to locate the occurrences of $maxRot(\alpha) = maxRot(val(z))$ and $maxRot(\beta) = maxRot(val^R(z))$. Then *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 $package(x_1, y_1, I_1) \cup package(x_2, y_2, I_2)$ are computed as described in the claim in Lemma 2. □

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

Algorithm 1: Count-Squares(T)

---

foreach $(T, r) \in \mathit{CDecomp}(\boldsymbol{T})$ **do**
  $\mathit{Semiruns} := \mathit{Semiruns}(T, r)$ /* Lemma 4 */
  Transform *Semiruns* into a set of packages in T /* Lemma 6 */
  Insert these packages to the set *Packages*

Compute disjoint representation of *Packages*
return $|\mathrm{sq}(\boldsymbol{T})|$ as the total length of intervals in *Packages*

The complexity of the **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. We consider each pair $(T, r) \in \mathit{CDecomp}(\mathbf{T})$ separately. Let m be the number of nodes of T. We will show that the computations for (T, r) can be performed in $O(m \log m)$ time. The total size of trees in the centroid decomposition of $\mathbf{T}$, $\sum_i m_i$, is $O(n \log n)$. This will conclude an $O(\sum_i m_i \log m_i) = O(\sum_i m_i \log n) = O(n \log^2 n)$ time algorithm.

First we build the rooted tree T_r and for this tree build the data structures for DBF, *dist* and *jump* queries and the *SHIFT* and 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 $\mathit{Semiruns}(T, r)$ in $O(m)$ time. By Lemma 6, the semiruns can be transformed to a cyclic representation of squares in T of size $O(m)$. This requires $O(m \log m)$ time.

Finally we compute a disjoint cyclic representation of all squares in $\mathbf{T}$ (that is, across all pairs (T, r)). For this, we group packages (x, y, I) in the cyclic representation according to $\mathit{val}(x, y)$, which is done by sorting them using Fact 3 for the comparison criterion. This takes $O(n \log^2 n)$ time. Afterwards in each group by elementary computations we turn a union of arbitrary cyclic intervals into a union of pairwise disjoint intervals. This requires sorting intervals, which is done simultaneously for all packages, so that the running time of this final phase is $O(n \log n)$. $\square$

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 $\mathit{val}(v) = \mathit{val}(w)$ implies that $v = w$. T_r is *semideterministic* if $\mathit{val}(v) = \mathit{val}(w)$

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

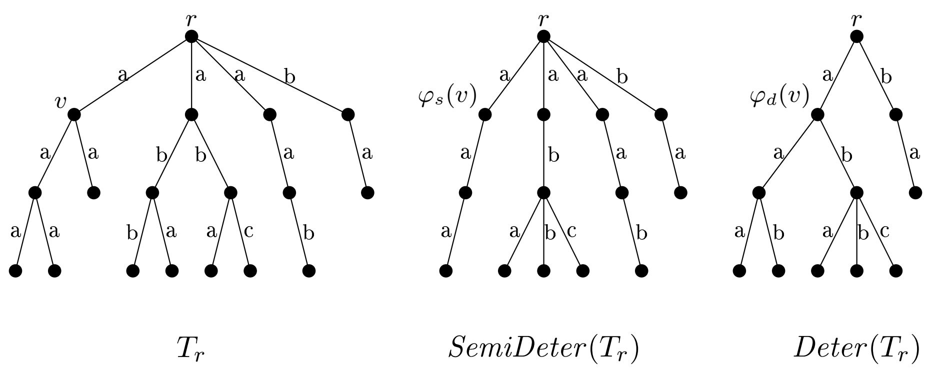


Figure 6: Different types of tree determinization.

Fact 4. *Let T_r be a rooted tree of n nodes. Then $Deter(T_r)$ and $SemiDeter(T_r)$ together with the corresponding pairs of functions φ_d, ψ_d and φ_s, ψ_s can be computed in $O(n)$ time.*

Proof. We first show how to compute the determinized tree $Deter(T_r)$. For each node v of T_r we compute the node $\varphi_d[v]$ of $Deter(T_r)$, corresponding to v in the determinized tree. We also compute an auxiliary table $children[v]$ for each node v in $Deter(T_r)$ containing the list of edges going down from v in $Deter(T_r)$, 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
$\varphi_d[r] := r$
foreach $(c, u, v) \in E$ **do** /* $c \in \Sigma$, $u, v \in T$ */
    if $(c, w) = last\_element(children[\varphi_d[u]])$ *for some* w **then**
        /* if (c, w) is in the list, it must be the last element */
        $\varphi_d[v] := w$
    else
        $\varphi_d[v] := v$
        $append(children[\varphi_d[u]], (c, \varphi_d[v]))$

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. □

Observation 4. *Note that φ_s and ψ_s preserve the values of paths going through r; this property does not hold for φ_d and ψ_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].

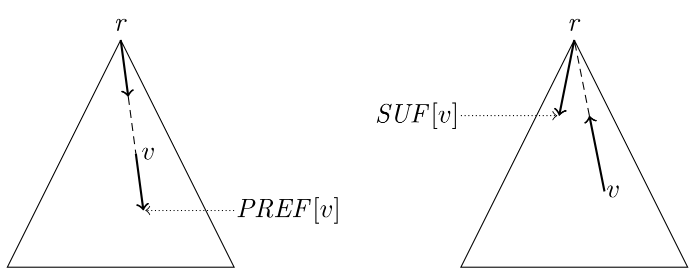


Figure 7: (a) $PREF[v]$; (b) $SUF[v]$.

- **[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).

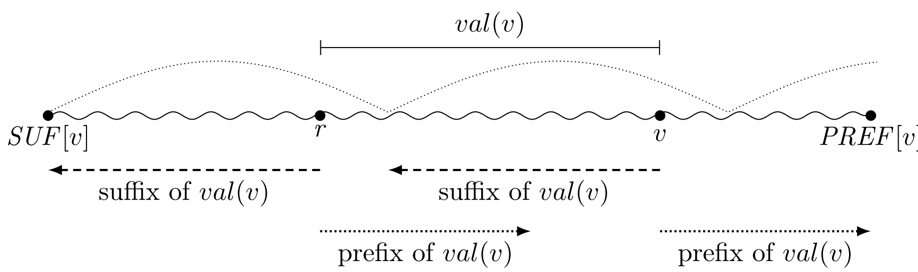


Figure 8: The semirun $(SUF[v], PREF[v], |val(v)|)$.

Observation 5. *Consider $v \neq r$. Then*

$$semirun(r, v) = (SUF[v], PREF[v], dist(v))$$

provided that the semirun exists.

Proof (of Lemma 4). Let T be a tree and r be one of its nodes. Let T_r be a copy of T rooted in r and let $T'_r = SemiDeter(T_r)$. 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 $Semiruns(T'_r, r)$ and returns the set:

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

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

Algorithm 3: Compute $Semiruns(T, r)$

$S := \emptyset$; $T'_r := SemiDeter(T_r)$
Compute the tables *PREF*, *SUF* for T'_r
foreach $v \in T'_r \setminus \{r\}$ **do**
  $x := PREF[v]$; $y := SUF[v]$
  if $dist(x, y) \geq 2 \cdot dist(v)$ **then**
    $S := S \cup \{(\psi_s(y),\ \psi_s(x),\ dist(v))\}$
return S

This completes the proof of Lemma 4 provided that we have the *PREF* and *SUF* tables. □

6. Computation of *PREF*

The *PREF* and *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 *PREF* table we generalize the results of Simon [18] originally developed for string pattern matching automata. For the *SUF* table, we use the suffix tree of a tree, a concept introduced by Kosaraju [19] for tree pattern matching.

Let T_r be a rooted semideterministic tree. We compute a slightly modified array $PREF'$ that allows for an overlap of the considered paths. More formally, for a node $v \neq r$, we define $PREF'[v]$ as the lowest node x in the subtree rooted at v such that $val(v,x)$ is a prefix of $val(x)$. Note that having computed $PREF'$, we can obtain *PREF* by truncating the result so that the paths do not overlap. This can be implemented with a single *jump* query.

Note that $PREF'[v]$ 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.

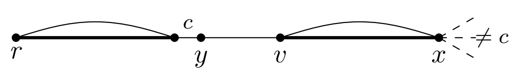


Figure 9: $PREF'[v] = x$ if and only if $val(v,x)c$ is a border of $val(x)c$ and no edge labeled with c leaves x.

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 $c \in \Sigma \setminus next(x)$.*

Proof. Note that in a deterministic tree $PREF'[v]$ can be (inefficiently) computed by the following procedure. Start with $z := r$ and $x := v$. Let a be the first letter of $val(z,x)$. If $a \in next(x)$, move x and z 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. $\square$

Let us define a *transition function* π, so that for a node x of T_r and $c \in \Sigma$, $\pi(x,c)$ is 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 π 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, see Fig. 10. 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.

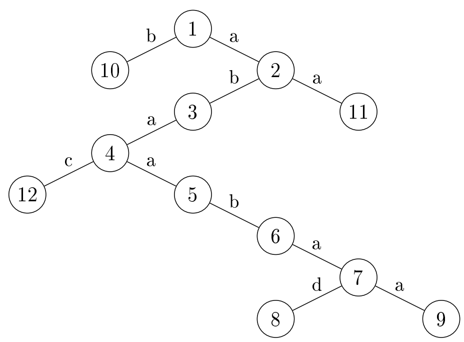


Figure 10: We have: $\pi[4] = \{(a, 2),\ (b, 3)\}$, $\pi[7] = \{(a, 5),\ (b, 3)\}$, $P[7] = 4$, $PREF'[3] = 4$, $PREF'[4] = 7$.

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 $c \in next(x)$ 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 $c \notin next(x)$. Let v be the only ancestor of x such that $dist(v, x) = dist(y) - 1$. 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 $c \notin next(x)$. This concludes the proof of the fact. $\square$

Before we present an algorithm computing π 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_r let $children[x]$ be a dictionary list mapping $c \in next(x)$ 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 π and P tables for T_r can be computed in $O(n)$ time.*

PROOF. In the algorithm we extensively use the dictionary lists $children[x]$. The transition lists π that we compute are also dictionary lists. For two dictionary lists L_1, L_2 indexed by Σ we define $L = merge(L_1, L_2)$ 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 $merge(L_1, L_2)$ is proportional to the total size of both lists.

Algorithm 4: Compute the π and P tables for T_r

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

Algorithm 4 computes the π 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 π 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 $children[x]$, which sums up to $O(n)$ over all nodes x. $\square$

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

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

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

Algorithm 5: Compute $PREF'$ for T_r

```
foreach x ∈ T_r \ {r} (in preorder) do
    foreach (c, (y, w)) ∈ merge(π[x], children[x]) do
        if w = nil and y ≠ nil then
            while y ≠ r do
                v := jump(x, r, dist(y) − 1)
                PREF'[v] := x
                y := P[y]
```

Let n be the number of nodes of T_r. For each node v of T_r we perform the assignment $PREF'[v] := x$ 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 π and *children* tables, which is also $O(n)$. $\square$

Lemma 10 concludes the "*PREF*" part of Lemma 4.

7. Computation of *SUF*

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

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

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

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

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

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

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

Unfortunately, the size of $\mathcal{T}$ 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 tree 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 $\mathcal{T}$ 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 $\mathcal{T}$ which yields a linear time algorithm constructing the SUF' table for T_r 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 $maxSuf(ww)$, 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 $maxSuf(w)$.*

Before we proceed, 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 $maxSuf(w)$ then i is not a square-center.*

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

Lemma 13 (Redundancy Lemma). *If u, v are borders of $maxSuf(w)$ such that $|u| < |v| \le 2|u|$ 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 $|v| - |u| \le |u|$. 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. $\square$

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

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 $Cand(w)$ and $Cand(w^R)$ we can compute $Cand(wa)$ and $Cand((wa)^R)$ in $O(\log|w|)$ time.*

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

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

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

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

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

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

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

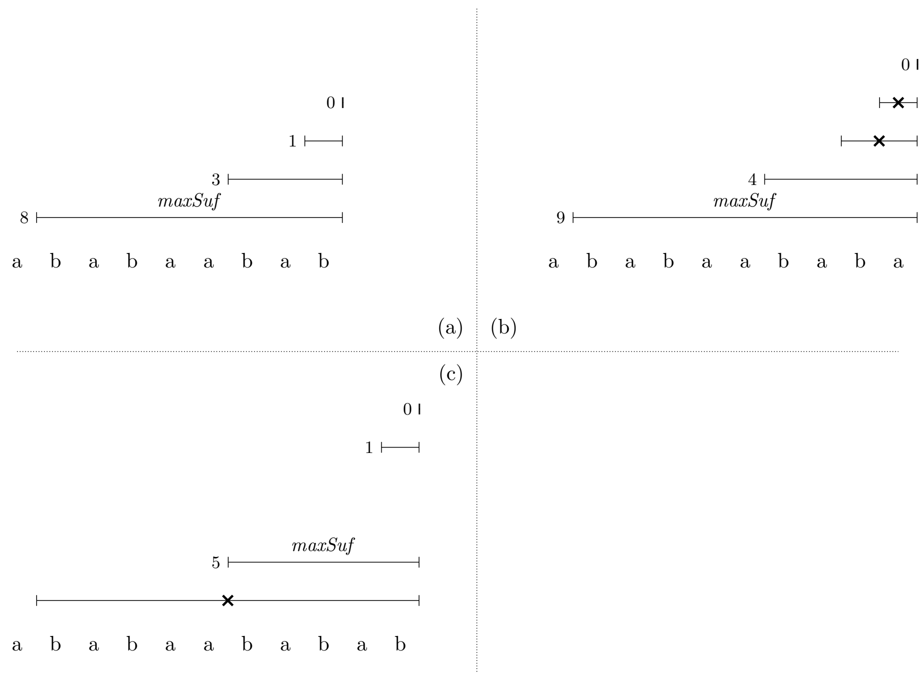


Figure 11: The computation of *Cand* sets as in the proof of Lemma 14. (a) *Cand* set for the word *ababaabab*. (b) *Cand* set for the word *ababaababa*, the suffixes of length 1 and 2 are removed due to point 4 of the procedure. (c) *Cand* set for the word *ababaababab*, here *maxSuf* changes and the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

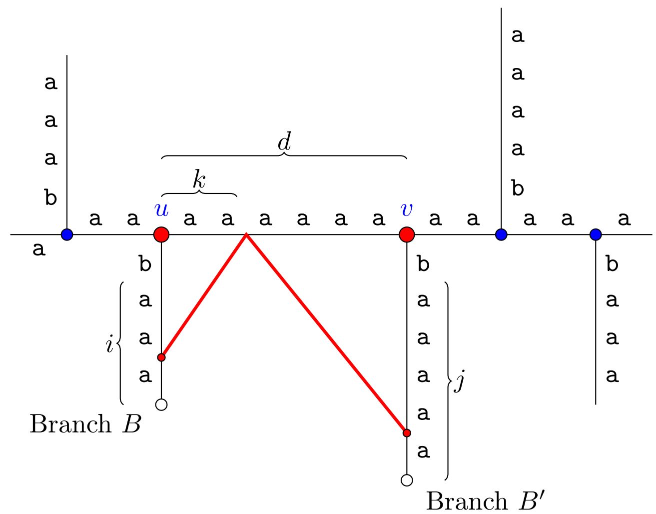


Figure 12: An essential part of a comb corresponding to nodes u, v on the spine consists of edges between them and outgoing branches starting with the letter b. The set of non-unary squares generated by this part equals $\{a^k b a^k a^{d-k} b a^{d-k} \ : \ k \in I(u,v)\}$, where $d = 6$ and $I(u,v) = [1,3]$.

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* $\mathrm{sq}(T)$ *can be computed in linear time.*

PROOF. We say that a pair u, v of nodes on the spine is *admissible* iff $d \le i + j$ where $d = dist(u, v)$ and i, j are numbers of a's on the branches attached to u and v. When finding all admissible pairs we can consider 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 package of squares generated by this pair, see Fig. 12, corresponds to an interval. These packages (for distinct pairs) are not necessarily disjoint. However, if for each d we find the union of intervals, we obtain a representation with disjoint packages. This can be done for all d simultaneously in linear time. We sum the numbers for each group and get the final result. □

Remark 1. *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* $\{a^k b a^k a^{d-k} b a^{d-k} \ : \ k \in [l, r]\}$.

10. Conclusions

The main result of this work is an $O(n \log^2 n)$ time algorithm computing the number of distinct squares in a labeled tree. The algorithm uses a compact representation of the set of all squares of $O(n \log n)$ size. An interesting open problem is whether there exists a faster solution to this problem, e.g. in $O(n \log n)$ time. The bottleneck of the approach presented in this paper (see the proof of Theorem 7) is the $O(n \log n)$ computation of shift tables and $O(n \log n)$ sorting of packages employing a Dictionary of Basic Factors in the comparison criterion.

References

[1] T. Kociumaka, J. Pachocki, J. Radoszewski, W. Rytter, T. Waleń, Efficient counting of square substrings in a tree, in: K.-M. Chao, T.-S. Hsu, D.-T. Lee (Eds.), ISAAC, Vol. 7676 of Lecture Notes in Computer Science, Springer, 2012, pp. 207–216.

[2] M. Crochemore, L. Ilie, W. Rytter, Repetitions in strings: Algorithms and combinatorics, Theor. Comput. Sci. 410 (50) (2009) 5227–5235.

[3] B. Bresar, J. Grytczuk, S. Klavzar, S. Niwczyk, I. Peterin, Nonrepetitive colorings of trees, Discrete Mathematics 307 (2) (2007) 163–172.

[4] S. Czerwiński, J. Grytczuk, Nonrepetitive colorings of graphs, Electronic Notes in Discrete Mathematics 28 (2007) 453–459.

[5] F. Blanchet-Sadri, R. Mercas, G. Scott, Counting distinct squares in partial words, Acta Cybern. 19 (2) (2009) 465–477.

[6] J. Grytczuk, J. Przybyło, X. Zhu, Nonrepetitive list colourings of paths, Random Struct. Algorithms 38 (1-2) (2011) 162–173.

[7] M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, J. Radoszewski, W. Rytter, W. Tyczyński, T. Waleń, The maximum number of squares in a tree, in: J. Kärkkäinen, J. Stoye (Eds.), CPM, Vol. 7354 of Lecture Notes in Computer Science, Springer, 2012, pp. 27–40.

[8] A. S. Fraenkel, J. Simpson, How many squares can a string contain?, J. of Combinatorial Theory Series A 82 (1998) 112–120.

[9] L. Ilie, A note on the number of squares in a word, Theor. Comput. Sci. 380 (3) (2007) 373–376.

[10] L. Ilie, A simple proof that a word of length n has at most $2n$ distinct squares, J. Comb. Theory, Ser. A 112 (1) (2005) 163–164.

[11] D. Gusfield, J. Stoye, Linear time algorithms for finding and representing all the tandem repeats in a string, J. Comput. Syst. Sci. 69 (4) (2004) 525–546.

[12] M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, T. Waleń, Extracting powers and periods in a string from its runs structure, in: E. Chávez, S. Lonardi (Eds.), SPIRE, Vol. 6393 of Lecture Notes in Computer Science, Springer, 2010, pp. 258–269.

[13] F. Harary, Graph Theory, Reading, MA: Addison-Wesley, 1994.

[14] D. Harel, R. E. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM J. Comput. 13 (2) (1984) 338–355.

[15] M. A. Bender, M. Farach-Colton, The level ancestor problem simplified, Theor. Comput. Sci. 321 (1) (2004) 5–12.

[16] M. Crochemore, W. Rytter, Jewels of Stringology, World Scientific, 2003.

[17] M. G. Main, R. J. Lorentz, An $O(n \log n)$ algorithm for finding all repetitions in a string, J. Algorithms 5 (3) (1984) 422–432.

[18] I. Simon, String matching algorithms and automata, in: J. Karhumäki, H. A. Maurer, G. Rozenberg (Eds.), Results and Trends in Theoretical Computer Science, Vol. 812 of LNCS, Springer, 1994, pp. 386–395.

[19] S. R. Kosaraju, Efficient tree pattern matching (preliminary version), in: FOCS, IEEE Computer Society, 1989, pp. 178–183.

[20] T. Shibuya, Constructing the suffix tree of a tree with a large alphabet, in: A. Aggarwal, C. P. Rangan (Eds.), ISAAC, Vol. 1741 of Lecture Notes in Computer Science, Springer, 1999, pp. 225–236.