Efficient Counting of Square Substrings in a Tree

Tomasz Kociumaka[1], Jakub Pachocki[1], Jakub Radoszewski[1],
Wojciech Rytter[1,2], and Tomasz Waleń[3,1]

[1] Dept. of Mathematics, Computer Science and Mechanics,
University of Warsaw, Warsaw, Poland
[kociumaka,jrad,pachocki,rytter,walen]@mimuw.edu.pl
[2] Dept. of Math. and Informatics,
Copernicus University, Toruń, Poland
[3] Laboratory 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 labeled trees. There are two main obstacles to overcome. The first one is that the number of such squares is $\Omega(n^{4/3})$, see Crochemore et al, 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

Various types of repetitions play an important role in combinatorics on words with particular applications in pattern matching, text compression, computational biology etc., see [3]. 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 [2]. Recently it has been shown that a tree with n nodes can contain $\Theta(n^{4/3})$ distinct squares, see [4], while the number of distinct squares in a string of length n does not exceed $2n$, as shown in [7]. This paper can be viewed as an algorithmic continuation of [4].

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 [8] or runs [5] are known.

Assume we have a tree T whose edges are labeled with symbols from an integer alphabet Σ. If u and v are two nodes of T, then by $val(u, v)$ we denote the sequence of labels of edges on the path from u to v (denoted as $u \rightsquigarrow v$). We call $val(u, v)$ a *substring* of T. (Note that a substring is a string, not a path.) Also let $dist(u, v) = |val(u, v)|$. Fig. 1 describes square substrings in a sample tree. We consider only simple paths: this means that vertices of a path do not repeat. Denote by $sq(T)$ the set of different square substrings in T. Our main result is computing $|sq(T)|$ in $O(n \log^2 n)$ time.

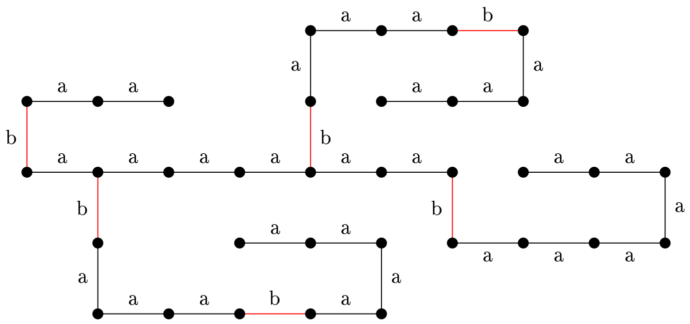


Fig. 1: We have here $|sq(T)| = 31$. There are 10 groups of cyclically equivalent squares, the representatives (maximal cyclic rotation of a square half) are: a, a^2, a^3, ba, ba^2, ba^3, ba^4, ba^5, ba^6, $(ba^3)^2$. For example, the equivalence class of $u^2 = (ba^6)^2$ contains the strings $rot(u, q)^2$ for $q \in [0, 3] \cup [5, 6]$, this is a single cyclic interval modulo 7 (see Section 3).

2 Algorithmic toolbox for trees

In this section we recall several well known concepts and apply them to design algorithms and data structures for labeled trees.

Navigation in trees. 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 queries can be answered in $O(1)$ time [9,1]. These queries let us efficiently navigate also in unrooted trees. For this purpose, we root the tree in an arbitrary node and split each path $x \rightsquigarrow y$ in $LCA(x, y)$. This way we obtain the result:

Fact 1 *Let T be a tree with n nodes. After $O(n)$ time preprocessing we 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 a positive integer $d \leq dist(x, y)$ compute $jump(x, y, d)$ — the node z on the path $x \rightsquigarrow y$ with $dist(x, z) = d$.

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 [6]. The DBF can be extended to arbitrary labeled trees. The proof of the following (nontrivial) fact is presented in the appendix.

Fact 2 *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).*

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, $CDecomp(T)$, is defined recursively:

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

The centroid of a tree can be computed in $O(n)$ time. The recursive definition of $CDecomp(T)$ implies a bound on its total size.

Fact 3 *Let T be a tree with n nodes. The total size of all subtrees in $CDecomp(T)$ is $O(n \log n)$. The decomposition $CDecomp(T)$ can be computed in $O(n \log n)$ time.*

Determinization. Let T be a tree rooted in R. We write $val(v)$ instead of $val(R, v)$, $val^R(v)$ instead of $val(v, R)$ and $dist(v)$ instead of $dist(R, v)$.

We say that T is *deterministic* if $val(v) = val(w)$ implies that $v = w$. We say that T is *semideterministic* if $val(v) = val(w)$ implies that $v = w$ or $R \rightsquigarrow v$ and $R \rightsquigarrow w$ are disjoint except R. Hence, T is semideterministic if it is "deterministic anywhere except for the root".

For an arbitrary tree T an "equivalent" deterministic tree $dtr(T)$ can be obtained by identifying nodes v, w if $val(v) = val(w)$. If we perform such identification only when the paths $R \rightsquigarrow v$, $R \rightsquigarrow w$ share the first edge, we obtain a semideterministic tree $semidtr(T)$. This way we also obtain functions φ mapping nodes of T to corresponding nodes in $dtr(T)$ (in $semidtr(T)$ respectively). Additionally we define $\psi(v)$ as an arbitrary element of $\varphi^{-1}(v)$ for $v \in dtr(T)$ ($v \in semidtr(T)$ respectively). A linear time implementation of these constructions is given in the appendix.

Note that φ and ψ for $semidtr(T)$ preserve the values of paths going through R; this property does not hold for $dtr(T)$.

3 Compact representations of sets of squares

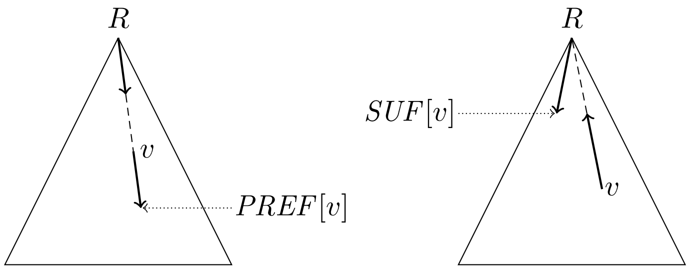
For a string u, by $rot(u)$ we denote the string u with its first letter moved to its end. For an integer q, by $rot(u, q)$ we denote $rot^q(u)$, i.e., the result of q iterations of the rot operation on the string u. If $v = rot(u, c)$ then u and v are called cyclically equivalent, we also say that v is a cyclic rotation of u and vice-versa. By $maxRot(u)$ we denote the lexicographically maximal cyclic rotation of u. 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 $dist(x, y)$. Define a *package* as a set of cyclically equivalent squares:

$$package(x, y, I) = \{rot(val(x, y), q)^2 : q \in I\}.$$

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.

Anchored squares. 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 $sq(T, v)$ we denote the set of squares anchored in v. Assume that T is rooted in R and let $v \neq R$ be a node of T with $dist(v) = p$. By $sq(T, R, v)$ we denote the set of squares of length $2p$ that have an occurrence passing through both R and v. Note that each path of length $2p$ passing through R contains a node v with $dist(v) = p$. Hence $sq(T, R)$ is the sum of $sq(T, R, v)$ over all nodes $v \neq R$.

We introduce two tables, defined for all $v \neq R$, similar to the tables used in Main-Lorentz square-reporting algorithm for strings [11]. In Section 5 we sketch algorithms computing these tables in linear time (for $PREF$ under additional assumption that the tree is semideterministic).



- **[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 figure above.
- **[Suffix table]** $SUF[v]$ is a lowest node x in T such that $val(x)$ is a prefix of $val^R(v)$ and $LCA(v, x) = R$.

We say that a string $s = s_1 \dots s_k$ has a period p if $s_i = s_{i+p}$ for all $i = 1, \dots, k-p+1$. Let x and y be nodes of T. A triple (x, y, p) is called a *semirun* if $val(x, y)$ has a period p and $dist(x, y) \geq 2p$. All substrings of $x \rightsquigarrow y$ and $y \rightsquigarrow x$ of length $2p$ are squares. We say that these squares are *induced* by the semirun. Let us fix $v \neq R$ with $dist(v) = p$. Note that $val(PREF[v], SUF[v])$ is periodic with period p. By the definitions of $PREF[v]$ and $SUF[v]$, if $dist(PREF[v], SUF[v]) < 2p$ then $sq(T, R, v) = \emptyset$. Otherwise $(PREF[v], SUF[v], p)$ is a semirun *anchored* in R and the set of squares it induces is exactly $sq(T, R, v)$, see also Figure 2.

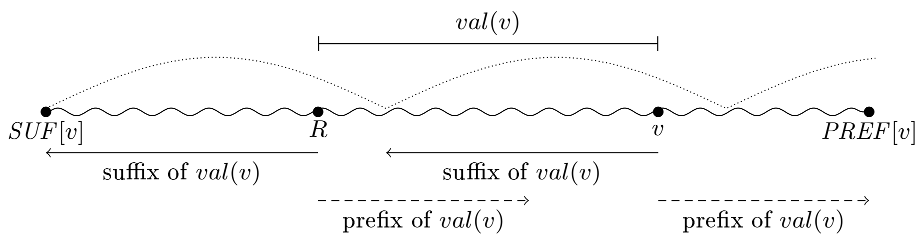


Fig. 2: The semirun $(SUF[v], PREF[v], |val(v)|)$ induces $sq(T, R, v)$.

For a set of semiruns S, by $sq(S)$ we denote the set of squares induced by at least one semirun in S. The following lemma summarizes the discussion on semiruns.

Lemma 1. *Let T be a tree of size n rooted in R. There exists a family S of $O(n)$ semiruns anchored in R such that $sq(S) = sq(T, R)$.*

Packages and semiruns. 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). The key tools for performing this translation are the following two tables defined for any node v of T.

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

In Section 5 we sketch algorithms computing these tables for a tree of n nodes in $O(n \log n)$ time.

Using these tables and the *jump* queries (Fact 1) we compute the cyclic representation of the set of squares induced by a family of semiruns.

Lemma 2. *Let T be tree of size n rooted in R and let S be a family of semiruns (x, y, p) anchored in R. There exists a cyclic representation of the set of squares induced by S that contains $O(|S|)$ packages and can be computed in $O(n \log n + |S|)$ time.*

Proof. Let $(x, y, p) \in S$. We have $LCA(x, y) = R$ and $dist(x, y) \geq 2p$, consequently there exists a node z on $x \rightsquigarrow y$ such that $dist(z) = p$ and squares induced by (x, y, p) are all cyclic rotations of $val(z)^2$ and $val^R(z)^2$. All cyclic rotations of $val(z)$ and $val^R(z)$ occur on the path $x \rightsquigarrow y$. The $SHIFT$ and $SHIFT^R$ tables can be used to locate the occurrences of $maxRot(val(z))$ and $maxRot(val^R(z))$. 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. This way we also obtain the cyclic intervals I_1 and I_2 that represent the set of squares induced by (x, y, p) as $package(x_1, y_1, I_1)$ and $package(x_2, y_2, I_2)$. □

The set of all squares. As a consequence of Lemmas 1 and 2 and Fact 3 (centroid decomposition) we obtain the following combinatorial characterization of the set of squares in a tree:

Theorem 1. *Let $\boldsymbol{T}$ be a labeled tree with n nodes. There exists a cyclic representation of all squares in $\boldsymbol{T}$ of $O(n \log n)$ size.*

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

4 Main algorithm

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

Algorithm 1: Count-Squares($\mathbf{T}$)

Compute DBF and *jump* data structure for $\mathbf{T}$ (Fact 1 and 2)
foreach $(T, R) \in CDecomp(\boldsymbol{T})$ **do**
    $Semiruns := semiruns(T, R)$
    Transform $Semiruns$ into a set of packages in T (Lemma 2)
    Insert these packages to the set $Packages$

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

Computing semiruns. Let $T' = \mathit{semidtr}(T)$. The following algorithm computes the set $S = \{(\psi(x), \psi(y), p) : (x, y, p) \in \mathit{semiruns}(T', R)\}$. Since $\mathit{sq}(T, R) = \mathit{sq}(T', R)$, this set of semiruns induces $\mathit{sq}(T, R)$.

Algorithm 2: Compute $\mathit{semiruns}(T, R)$

$S := \emptyset$; $T' := \mathit{semidtr}(T)$
Compute the tables $\mathit{PREF}(T'), \mathit{SUF}(T')$
foreach $v \in T' \setminus \{R\}$ **do**
  $x := \mathit{PREF}[v]$; $y := \mathit{SUF}[v]$
  if $\mathit{dist}(x, y) \geq 2 \cdot \mathit{dist}(v)$ **then**
    $S := S \cup \{(\psi(x),\ \psi(y),\ \mathit{dist}(v))\}$
return S

Computing a disjoint representation of ***Packages*****.** In this phase we compute a compact representation of distinct squares. For this, we group packages (x, y, I) according to $\mathit{val}(x, y)$, which is done by sorting them using Fact 2 to implement the comparison criterion efficiently. 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.

In total, precomputing DBF, *jump* and *CDecomp* and computing semiruns takes $O(n \log n)$ time, whereas transforming semiruns to packages and computing a disjoint representation of packages takes $O(n \log^2 n)$ time:

Theorem 2. *The number of distinct square substrings in an (unrooted) tree with n nodes can be found in $O(n \log^2 n)$ time.*

5 Construction of the basic tables

Computation of ***PREF*****.** 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 $\mathit{PREF}'[v]$ as the lowest node x in the subtree rooted in v such that $\mathit{val}(v, x)$ is a prefix of $\mathit{val}(x)$. Note that having computed PREF', we can obtain PREF by truncating the result so that paths do not overlap. This can be implemented with a single *jump* query.

Note that $\mathit{PREF}'[v]$ depends only on the path $R \rightsquigarrow v$ 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 is deterministic.

Recall that a border of a string w is a string that is both a prefix and a suffix of w. The $PREF$ function for strings is closely related to borders, see [6]. This is inherited by $PREF'$ for deterministic trees, see Figure 3.

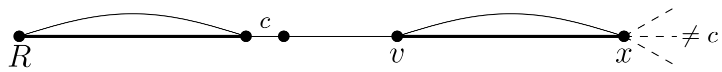


Fig. 3: $PREF'[v] = x$ if and only if $val(v, x)c$ is a border of $val(x)c$ and no edge labeled with c leaves x.

For a node x of T 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 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. For $x \neq R$ we set $P[x]$ as the node y such that $val(y)$ is the longest border of $val(x)$. The following lemma generalizes the results of [13] and gives the crucial properties of essential transitions. Its proof can be found in the appendix.

Lemma 3. *Let T be a deterministic tree of n nodes. There are no more than $2n - 1$ essential transitions in T. Moreover, the π and P tables can be computed in $O(n)$ time.*

In the algorithm computing the $PREF'$ table, for each x we find all nodes v such that $PREF[v] = x$. This is done by iterating the P table starting from $\pi(x, c)$, see also Fig. 3. Details can be found in the appendix.

Lemma 4. *For a deterministic tree T, the table $PREF'(T)$ can be computed in linear time.*

Computation of SUF. Let T be a deterministic tree rooted in R and $v \neq R$ be a node of T. We define $SUF'[v]$ as the lowest node x of T 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 T is deterministic. A technical proof of the following lemma in given in the appendix.

Lemma 5. *Let T be an arbitrary rooted tree of n nodes. The $SUF(T)$ table can be computed in $O(n)$ time from $SUF'(dtr(T))$.*

Observe that *tries* are exactly the deterministic rooted trees. Let $S_1 = \{val(x) : x \in T\}$ and $S_2 = \{val^R(x) : x \in T\}$. Assume we have constructed a trie $\mathcal{T}$ of all the strings $S_1 \cup S_2$ and that we store the pointers to nodes in $\mathcal{T}$ that correspond to elements of S_1 and S_2. Then for any $v \in T$, $SUF'[v]$ corresponds to the lowest ancestor of $val^R(v)$ in $\mathcal{T}$ of the form $val(x)$. Such ancestors can be computed by a single top-bottom 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 nodes having at least two children. The trie of S_1 is exactly T, whereas the compacted trie of S_2 is known as a *suffix tree of the tree* T. This notion was introduced in [10] and a linear time construction algorithm for an integer alphabet was given in [12]. The compacted trie $\mathcal{T}$ can therefore be obtained by merging T with its suffix tree, i.e. identifying nodes of the same value. Since T 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 and consequently the following result:

Lemma 6. *The SUF table of a rooted tree can be computed in linear time.*

Computation of $SHIFT$ **and** $SHIFT^R$**.** The computation of maximal rotation (shift) of w is equivalent to finding $maxSuf(ww)$, see [6]. 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 *nonredundant*.

Observation 1 *(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 nonredundant suffix of w, then u is a prefix, and therefore a border, of $maxSuf(w)$.*

Lemma 7 (Redundancy Lemma). *If u, v are borders of $maxSuf(w)$ such that $|u| < |v| \leq 2|u|$ then u is a redundant suffix of w.*

Definition 1. *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 nonredundant suffixes of w and $|Cand(w)| \leq \max(1, \log|w| + 1)$.*

Lemma 8. *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.

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 $|u| < |v| \leq 2|u|$, 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 7 that the resulting set $Cand(wa)$ is a small candidate set. $Cand((wa)^R)$ is computed in a similar way. □

Lemma 9. *For a labeled rooted tree T the tables $SHIFT$ and $SHIFT^R$ can be computed in $O(n \log n)$ time.*

Proof. We traverse the tree in DFS order 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. □

References

1. M. A. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004.
2. B. Bresar, J. Grytczuk, S. Klavzar, S. Niwczyk, and I. Peterin. Nonrepetitive colorings of trees. *Discrete Mathematics*, 307(2):163–172, 2007.
3. M. Crochemore, L. Ilie, and W. Rytter. Repetitions in strings: Algorithms and combinatorics. *Theor. Comput. Sci.*, 410(50):5227–5235, 2009.
4. M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, J. Radoszewski, W. Rytter, W. Tyczyński, and T. Waleń. The maximum number of squares in a tree. In J. Kärkkäinen and J. Stoye, editors, *CPM*, volume 7354 of *Lecture Notes in Computer Science*, pages 27–40. Springer, 2012.
5. M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Waleń. Extracting powers and periods in a string from its runs structure. In E. Chávez and S. Lonardi, editors, *SPIRE*, volume 6393 of *Lecture Notes in Computer Science*, pages 258–269. Springer, 2010.
6. M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2003.
7. A. S. Fraenkel and J. Simpson. How many squares can a string contain? *J. of Combinatorial Theory Series A*, 82:112–120, 1998.
8. D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69(4):525–546, 2004.
9. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
10. S. R. Kosaraju. Efficient tree pattern matching (preliminary version). In *FOCS*, pages 178–183. IEEE Computer Society, 1989.
11. M. G. Main and R. J. Lorentz. An O(n log n) algorithm for finding all repetitions in a string. *J. Algorithms*, 5(3):422–432, 1984.
12. T. Shibuya. Constructing the suffix tree of a tree with a large alphabet. In A. Aggarwal and C. P. Rangan, editors, *ISAAC*, volume 1741 of *Lecture Notes in Computer Science*, pages 225–236. Springer, 1999.
13. I. Simon. String matching algorithms and automata. In J. Karhumäki, H. A. Maurer, and G. Rozenberg, editors, *Results and Trends in Theoretical Computer Science*, volume 812 of *LNCS*, pages 386–395. Springer, 1994.

Appendix

A1. Details of fast algorithm for path label comparison

Fact 2 *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' 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 od two 2^i and node $v \in T'$, 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.

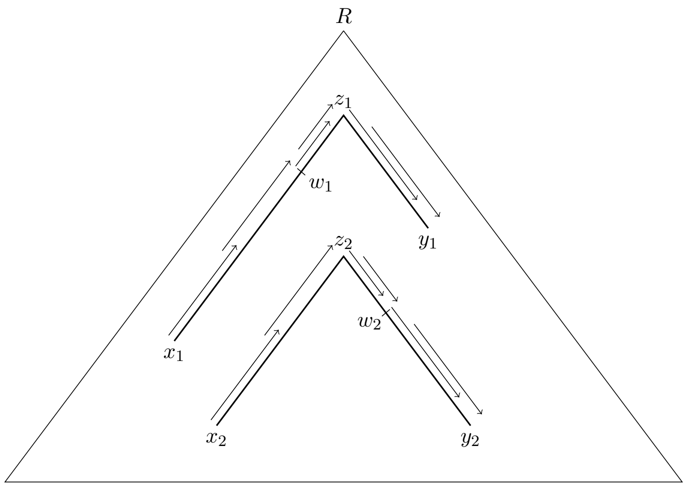


Fig. 4: Comparing $val(x_1, y_1)$ and $val(x_2, y_2)$ ($dist(x_1, y_1) = dist(x_2, y_2)$).

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. 4. For example, $val(x_1, w_1)$, with $d = dist(x_1, w_1)$, corresponds to:

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

whereas $val(z_1, y_1)$, with $d' = dist(z_1, y_1)$, corresponds to:

$$(id^R(jump(y_1, z_1, d' - 2^j), j),\ id^R(y_1, j)) \quad \text{for} \quad 2^j \leq d' < 2^{j+1}.$$

Thus the comparison of $val(x_1, y_1)$ and $val(x_2, y_2)$ reduces to a comparison of two 6-tuples that can be performed in $O(1)$ time. □

A2. Details of tree determinization

We show how to compute the determinized tree $dtr(T)$ for a given labeled tree T rooted in a node R. For a given node v of T we compute the node $\varphi[v]$ of $dtr(T)$, corresponding to v in the determinized tree. We also compute an auxiliary table $children[w]$ for each node w in $dtr(T)$, containing the list of edges going down from w in $dtr(T)$, sorted by the labels.

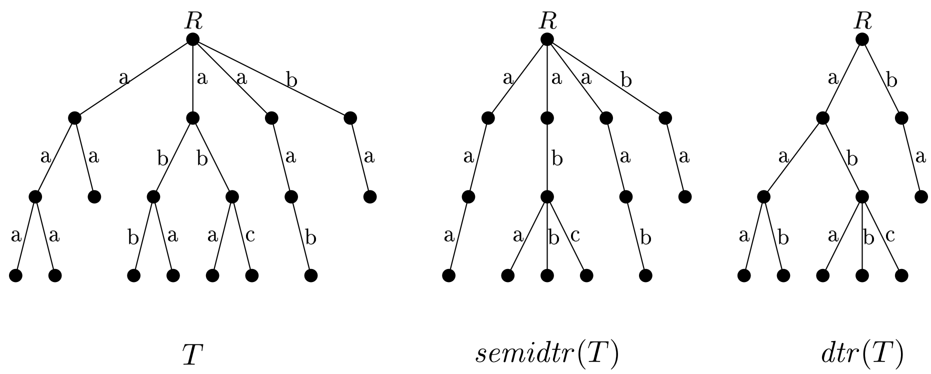


Fig. 5: Different types of tree determinization.

Counting sort can be employed for sorting the edges; consequently, the following Algorithm 3 works in linear time.

To compute $semidtr(T)$ it suffices to apply Algorithm 3 to all subtrees rooted in children of R.

Algorithm 3: Compute $dtr(T)$ **for** (T, R)

sort all edges in T by label and store them in E
stably sort E by the depth of the edges (with edges closest to the root first)
initialize *children* to be empty
$\varphi[R] := R$
foreach $(c, u, v) \in E$ **do**
  if $\exists_w (c, w) \in children[\varphi[u]]$ **then**
    $\{$ (c, w) must be the tail of the list $children[\varphi[u]]$ $\}$
    $\varphi[v] := w$
  else
    $\varphi[v] := v$
    $children[\varphi[u]] := children[\varphi[u]].\mathrm{append}(c, \varphi[v])$

A3. Details of computation of *PREF*

First, let us prove the following fact, see also Figure 3. Here $next(x)$ denotes the set of labels of edges leaving x.

Fact 4 *Let T be a deterministic tree rooted in R. Let $v \neq R$ be a node in T 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 $y := R$ and $x := v$. Let a be the first letter of $val(y, x)$. If $a \in next(x)$, 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. $\square$

Now we are ready for the proofs of Lemma 3 and Lemma 4. Each of the proofs actually contains a pseudocode of an algorithm computing the respective tables.

Lemma 3. *Let T be a deterministic tree of n nodes. There are no more than $2n - 1$ essential transitions in T. Moreover, the π and P tables can be computed in $O(n)$ time.*

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 transition 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 4, 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 combinatorial part of the lemma.

Before we present the 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 let $children[x]$ be a dictionary list mapping $c \in next(x)$ into the corresponding child nodes of x. We assume that we have such lists — actually our determinization algorithm produces trees with edges sorted by labels. The transition lists π 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) := []
foreach (c, w) ∈ children[R] do
    P[w] := R
foreach x ∈ T \ {R} (preorder) do
    y := P[x]
    y' := jump(y, x, 1)
    a := val(y, y')
    π[x] := []
    foreach (b, u) ∈ π[y] do
        if a ≠ b then
            π[x].append(b, u)
    π[x].insert(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. □

Lemma 4. *For a deterministic tree T, the table $PREF'(T)$ can be computed in linear time.*

Proof. Algorithm 5 uses the characterization of $PREF'$ given by Fact 4. It iterates over all borders of $val(x)c$. The longest one is found using the transition table π. The remaining ones are computed by iterating the border table P.

Let n be the number of nodes of T. For each v 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 $O(n)$. □

Algorithm 5: Compute $PREF'$ for (T, R)

```
foreach x ∈ T \ {R} (preorder) do
    foreach (b, (u, w)) ∈ merge(π[x], children[x]) do
        if w = nil ∧ u ≠ nil then
            while u ≠ R do
                v := jump(x, R, dist(u) − 1)
                PREF′[v] := x
                u := P[u]
```

A4. Details of computation of *SUF*

The only remaining detail is the following reduction:

Lemma 5. *Let T be an arbitrary rooted tree of n nodes. The table $SUF(T)$ can be computed in $O(n)$ time from $SUF'(dtr(T))$.*

Proof. Recall the φ function mapping a node of T to the corresponding node in $dtr(T)$. For a node $x \neq R$ of T let $subroot(x)$ be the child of R lying on the path $R \rightsquigarrow x$. For a node $y' \neq R$ of $dtr(T)$ let $subroots(y') = \{subroot(y) : y \in \varphi^{-1}(y')\}$. All the subroots can easily be precomputed in linear time. Moreover, together with $z \in subroots(y')$ we can store $y \in \varphi^{-1}(y')$ 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 $\varphi(y)$ is an ancestor of $y' = SUF'[\varphi(x)]$. Note that the *subroots* function is monotonic, so either $\varphi(y) = y'$ or $subroots(y') = \{subroot(x)\}$ and $\varphi(y)$ is the lowest ancestor of y' whose *subroots* set contains at least two elements. Such ancestors can be precomputed for all nodes of $dtr(T)$ by a single top-down tree traversal.

Once we know $z' = \varphi(y)$ 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 $z \in \varphi^{-1}(z')$ is the node associated with this subroot. $\square$

A5. Proof of Redundancy Lemma for *SHIFT* tables

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 5 *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. $\square$

Lemma 7. *If u, v are borders of $maxSuf(w)$ such that $|u| < |v| \leq 2|u|$ then u is a redundant suffix of w.*

Proof. Due to Fine & Wilf's periodicity lemma [6] 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 5, uz is not the maximal suffix of wz. $\square$