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] Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Warsaw, Poland
[kociumaka,jrad,pachocki,rytter,walen]@mimuw.edu.pl
[2] Faculty of Mathematics and Computer Science, Nicolaus 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. Crochemore et al. showed in 2012 that the number of such squares is bounded by $\Theta(n^{4/3})$. This is substantialy different from the case of classical strings, which admit only a linear number of distinct squares. We deal with this difficulty by introducing a compact representation of all squares (based on maximal cyclic shifts) that requires only $O(n \log n)$ space. The second obstacle is lack of adequate algorithmic tools for labeled trees. Consequently we develop several novel techniques, which form the most complex part of the paper. In particular we extend Imre Simon's implementation of the failure function 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 repetitions 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 at most $\Theta(n^{4/3})$ 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].

Despite the linear upper bound, enumerating squares in ordinary strings is already a difficult problem. Complex $O(n)$ time solutions 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 let $val(u, v)$ denote the sequence of labels of edges on the path from u to v (denoted as $u \rightsquigarrow v$). We

⋆ The author is supported by grant no. N206 566740 of the National Science Centre.

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 presents square substrings in a sample tree. We consider only simple paths, that is the 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.

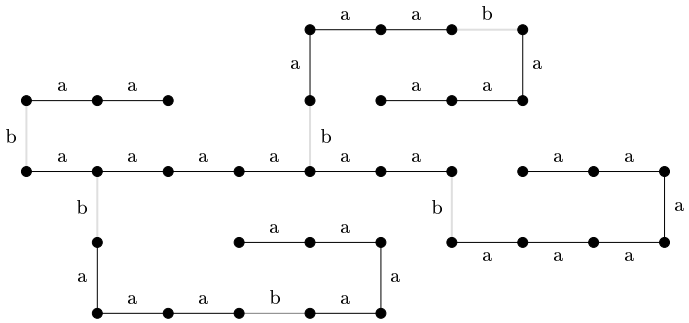


Fig. 1: We have here $|sq(T)| = 31$. There are 10 groups of cyclically equivalent squares, the representatives (maximal cyclic shift of of a square half) are: a, a^2, a^3, ba, ba^2, ba^3, ba^4, ba^5, ba^6, $(ba^3)^2$. For example, the equivalence class of $u^2 = (ba^6)^2$ contains the strings $rot(u,q)^2$ 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 apply several well known concepts 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 types of queries can be answered in $O(1)$ time [9,1]. We use them to 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 following 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 nonnegative 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. Due to the lack of space the proof of the following (nontrivial) fact will be presented in the full version of the paper.

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 with n nodes and $T_1, T_2, \ldots, T_k$ 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 each 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 an $O(n \log n)$ 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 at r. We write $val(v)$ instead of $val(r, v)$, $val^R(v)$ instead of $val(v, r)$ and $dist(v)$ instead of $dist(r, v)$.

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 $v = w$ unless $r \rightsquigarrow v$ and $r \rightsquigarrow w$ are disjoint except r. Informally, 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 and w if $val(v) = val(w)$. If we perform such identification only when the paths $r \rightsquigarrow v$ and $r \rightsquigarrow w$ share the first edge, we get 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). Note that φ and ψ for $semidtr(T)$ preserve the values of paths going through r; this property does not hold for $dtr(T)$.

The details of efficient implementation are left for the full version of the paper:

Fact 4 *Let T be a rooted tree with n nodes. The trees $dtr(T)$ and $semidtr(T)$ together with the corresponding pairs of functions φ and ψ can be computed in $O(n)$ time.*

3 Compact representations of sets of squares

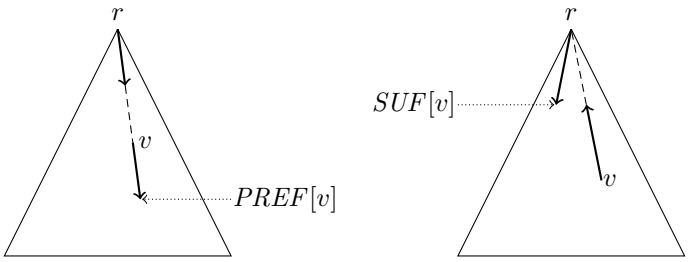
For a string u, let $rot(u)$ denote the string u with its first letter moved to the end. For an integer q, let $rot(u, q)$ denote $rot^q(u)$, i.e., the result of q iterations of the rot operation on the string u. If $v = rot(u, c)$ then u and v are called cyclically equivalent, we also say that v is a cyclic rotation of u and vice-versa. Let $maxRot(u)$ 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. Let $sq(T, v)$ denote the set of squares anchored in v. Assume that T is rooted at r and let $v \neq r$ be a node of T with $dist(v) = p$. Let $sq(T, r, v)$ 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 the Main-Lorenz square-reporting algorithm for strings [11]. In Section 5 we sketch algorithms computing these tables in linear time (for $PREF$ under the 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 $i = 1, \dots, k-p$. 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.

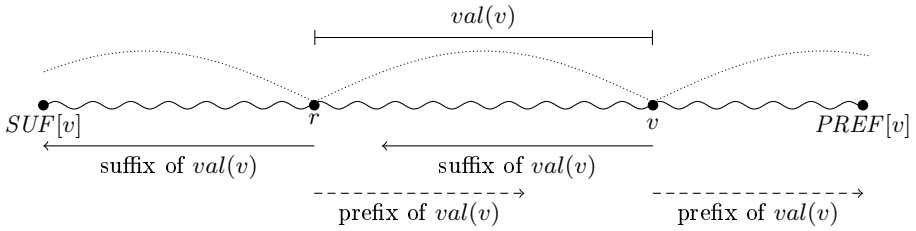


Fig. 2: The semirun $(SUF[v], PREF[v], |val(v)|)$ induces $sq(T, r, v)$.

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

Lemma 5. *Let T be a tree of size n rooted at 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 the smallest nonnegative integer r such that $rot(val(v), r) = maxRot(val(v))$.
2. **[Reversed Shift Table]** $SHIFT^R[v]$ is the smallest nonnegative 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 with 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 6. *Let T a be tree of size n rooted at 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 all squares induced by (x, y, p) are cyclic rotations of $val(z)^2$ and $val^R(z)^2$. Observe that $maxRot(val(z))$ and $maxRot(val^R(z))$, as any cyclic rotation of $val(z)$ or $val^R(z)$, occur on the path $x \rightsquigarrow y$. The *SHIFT* and $SHIFT^R$ tables can be used to locate these occurrences, then *jump* queries allow to find their exact endpoints, nodes x_1, y_1 and x_2, y_2 respectively. 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 5 and 6 and Fact 3 (centroid decomposition) we obtain the following combinatorial characterization of the set of squares in a tree:

Theorem 7. *Let* $\mathbf{T}$ *be a labeled tree with* n *nodes. There exists a cyclic representation of all squares in* $\mathbf{T}$ *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

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

Algorithm 1: Compute $semiruns(T, r)$

```
S := ∅;  T' := semidtr(T)
Compute the tables PREF(T'), SUF(T')
foreach v ∈ T' \ {r} do
    x := PREF[v];  y := SUF[v]
    if dist(x, y) ≥ 2 · dist(v) then
        S := S ∪ {(ψ(x), ψ(y), 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 $val(x, y)$, 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.

General structure of the algorithm. The structure is based on the centroid decomposition. 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 8. *The number of distinct square substrings in an (unrooted) tree with n nodes can be found in $O(n \log^2 n)$ time.*

Algorithm 2: Count-Squares(T)

```
Compute DBF and jump data structure for T;                    {Facts 1 and 2}
foreach (T, r) ∈ CDecomp(T) do
    Semiruns := semiruns(T, r)
    Transform Semiruns into a set of packages in T;           {Lemma 6}
    Insert these packages to the set Packages
Compute (interval) disjoint representation of Packages
return |sq(T)| as the total length of intervals in Packages
```

5 Construction of the basic tables

The *PREF* and *SUF* tables for ordinary strings are computed by a single simple algorithm, see [6]. 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 [13] originally developed for string pattern matching automata. For the *SUF* table, we use the suffix tree of a tree, originally designed by Kosaraju [10] for tree pattern matching.

5.1 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 $PREF'[v]$ 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 paths do not overlap. This can be implemented with a single *jump* query.

Observe that $PREF'[v]$ depends only on the path $r \rightsquigarrow v$ and the subtree rooted at v. Hence, instead of a single semideterministic tree with 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.

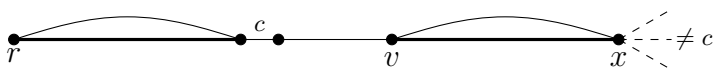


Fig. 3: $PREF'[v] = x$ if and only if $val(v, x)c$ is a border of $val(x)c$ and no edge labeled with c leaves x.

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. The proof is left for the full version of the paper.

Lemma 9. *Let T be a deterministic tree with 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.

Lemma 10. *For a deterministic tree T, the table $PREF'(T)$ can be computed in linear time.*

5.2 Computation of *SUF*

Let T be a deterministic tree rooted at 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. The technical proof of the following lemma is left for the full version of the paper.

Lemma 11. *Let T be an arbitrary rooted tree with 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 that we have constructed a trie $\mathcal{T}$ of all the strings $S_1 \cup S_2$ and that we store pointers to the 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 $\mathcal{T}$ which comes from S_1. Such ancestors can be computed for all nodes 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. Hence, we obtain 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 12. *The SUF table of a rooted tree can be computed in linear time.*

5.3 Computation of *SHIFT* and $SHIFT^R$

Recall that the maximal rotation of w corresponds to the maximal suffix of ww, see [6]. We develop algorithms based on this relation using a concept of *redundant* suffixes. A redundant suffix of w cannot become maximal under any extension of w, regardless of the direction we extend, see Observation 14.

Definition 13. *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 14 *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.*

In the following easy fact and subsequent lemmas we build tools, which allow to focus on a logarithmic number of suffixes, discarding others as redundant.

Fact 15 *If u is a nonredundant suffix of w, then u is a prefix, and therefore a border, of $maxSuf(w)$, the lexicographically maximum suffix of w.*

Lemma 16. *Let i be a position in a string w. Assume i is a square center, i.e. there exists a square factor of w whose second half starts at i. Then the suffix of w starting at i is redundant.*

Proof. Let $w = uxxv$, where $|ux| = i-1$. We need to show that xv is a redundant suffix of w. Let z be an arbitrary string. Consider three suffixes of wz: vz, xvz and $xxvz$. If $vz < xvz$, then $xvz < xxvz$, otherwise $xvz < vz$. Hence, for each z there exists a suffix of wz greater than xvz, which makes xv redundant. □

Lemma 17. *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, by Fact 16, u is redundant. □

Above lemmas do not give a full characterization of nonredundant suffixes, hence our algorithm maintains a carefully defined set that might be slightly larger.

Definition 18. *We call a set C a* small candidate set *for a string w if C is a subset of borders of $maxSuf(w)$, contains all nonredundant suffixes of w and $|C| \leq \max(1, \log|w| + 1)$. Any small candidate set for w is denoted by $Cand(w)$.*

Lemma 19. *Assume we are given a string w and we are able to compare factors of w in constant time. Then for any $a \in \Sigma$, given small candidate set $Cand(w)$ ($Cand(w^R)$) we can compute $Cand(wa)$ (resp. $Cand((wa)^R)$) 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 definitions of redundancy and small candidate set.
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 Observation 14, Fact 15 and Lemma 17 that the resulting set $Cand(wa)$ is a small candidate set. $Cand((wa)^R)$ is computed in a similar way. □

Lemma 20. *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 constant 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.