String Covers of a Tree Revisited

Łukasz Kondraciuk[0000−0002−7332−6533]

University of Warsaw, Poland
lk385775@students.mimuw.edu.pl

Abstract. We consider covering labeled trees with a collection of paths with the same string label, called a (string) cover of a tree. This problem was originated by Radoszewski et al. (SPIRE 2021), who show how to compute all covers of a directed rooted labeled tree in $O(n \log n / \log \log n)$ time and all covers of an undirected labeled tree in $O(n^2)$ time and space, or $O(n^2 \log n)$ time and $O(n)$-space. (Here n denotes the number of nodes of a given tree.) We improve those results by proposing a linear time algorithm for reporting all covers of a directed tree, and showing an $O(n^2)$ time and $O(n)$-space algorithm for computing undirected tree covers. Both algorithms assume that labeling characters come from an integer alphabet.

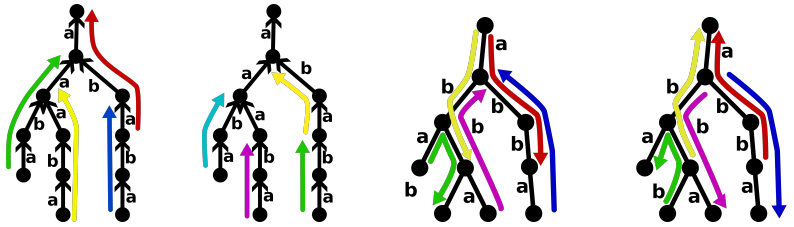
1 Introduction

String C is a cover of string S if every character of S belongs to at least one substring of S equal to C. Cover-related problems have been studied since at least 1990. Apostolico and Ehrenfeucht introduced this feature of a string in [2]. Apostolico, Farach, and Iliopoulos in [3] discovered an algorithm for checking if string S contains any covers, other than S, in $O(|S|)$ time. They called S *superprimitive* if it doesn't contain any cover other than itself. Dany Breslauer in [4] extended this algorithm to work online - it tests if each prefix of the input string is superprimitive as soon as the prefix is given. Please note that *cover* and *quasiperiod* are equivalent terms. However, it is not clear how to define periodicity when we switch from words to trees, thus later in this paper we will only use the term *cover*.

Moore and Smyth [15] were the first to discover a way to report all covers of a string S in $O(|S|)$ time – our algorithm and the previous algorithm [16] use their result as a starting point for the directed cover problem. Czajka and Radoszewski in [8] evaluated the practical performance of algorithms computing covers of strings. For a very recent survey on other variants of covers, see [14].

Let us consider a rooted tree T, consisting of n nodes. Each of its edges is labeled by a single character $\in \Sigma$. A simple directed path (let us denote it as p) is a non-repeating sequence of nodes. Each pair of consecutive nodes is connected by an edge. The first node of this sequence is a start point of p - let us denote it as s. The last node is an endpoint of p - let us denote it as e. We will define $s \to e$ as a sequence of nodes on a single path from s to e, equivalent to p. Any simple directed path of T is uniquely identified by a two-element tuple

(aka ordered pair) (startpoint, endpoint). The label of a directed simple path is a string constructed by concatenating characters on edges connecting consecutive nodes. String C is a cover of a tree T, if there exists a set of simple paths M, each of them having a label equal to C, so that each edge of T belongs to at least one path from M. We will consider two variants of this problem, which were first proposed and studied by Radoszewski et al. in [16]. Similar problems, involving labeled trees and computing their runs, powers, and palindromes, were extensively studied in [5], [7], [9], [10], [12], [13], [19], and most recently in [11].



(a) (b) (c) (d)

Fig. 1: a) and b): aba is a directed cover of this tree, ab is not: 3 edges cannot be covered by this string. c) and d): abb, $bba = (abb)^R$, and $ba = (ab)^R$ are undirected covers this tree (only aab and bba are visualized).

For the directed cover problem the considered tree is rooted at some node, and we add a restriction that all paths from M should be only going up. It means that for any $p \in M$ its endpoint is an ancestor of its startpoint. The algorithm presented in [16] runs in time $O(n \log n / \log \log n)$ time and $O(n)$-space, and requires the labeling character alphabet to be integer. Our improvement involves preprocessing the input tree by compacting branchless paths, which later bounds the total number of iterations of an inner loop of an algorithm, instead of utilizing a data structure for the dynamic marked ancestor problem. This gives us $O(n)$ time and space algorithm. Unfortunately, it still requires the labeling alphabet to be integer.

For the undirected cover problem, there are no more restrictions. Algorithms described in [16] were $O(n^2)$ time $O(n^2)$ space (divides paths of a tree between cover-candidates all at once), and $O(n^2 \log n)$ time $O(n)$-space (this one verifies only one cover-candidate at the time). Our algorithm will combine both ideas by, for one candidate at a time, finding a compressed set of paths that this candidate covers. This way we can achieve $O(n^2)$ time $O(n)$-space complexity.

Figure 1 shows examples of a directed and an undirected cover. Each cover will be reported by our algorithms as an ordered pair of nodes. (v, w) will be representing a string - label on a path $v \to w$. This way we can report all covers (we will later prove that in both variants there are at most $O(n)$ of them) in $o(n^2)$ time, even though their straightforward representation can be as large as

$\Theta(n^2)$. This is the case for instance when a given tree forms a simple path, and all of its edges are labeled by the same letter.

We provide reference Python implementations of the described algorithms.[1]

2 Preliminaries

A string S is a sequence of characters $S[1], S[2], ..., S[|S|] \in \Sigma$. A substring of S is any string of the form $S[i..j] = S[i], S[i+1], \ldots, S[j]$. If $i = 1$ ($j = |S|$), it is called a prefix (a suffix, respectively), $S^R = S[|S|], S[|S|-1], ..., S[1]$.

Let us consider any string S. A cover C is a substring of S, which occurs at some positions of S, and each letter of S is covered by at least one occurrence of C. Formally, string C is a cover of S, if there exists a set of positions $M \subseteq \{1, \ldots, |S| - |C| + 1\}$, such that for every $i \in M$, $S[i..(i + |C| - 1)] = C$ (M represents a – not necessarily proper – subset of occurrences of C), and for every $1 \leq i \leq |S|$, there exists $j \in M$, such that $i - |C| + 1 \leq j \leq i$ (every letter of S should be covered by an occurrence).

When we consider a rooted tree and algorithms processing it, it is helpful to define a few properties of a tree and its nodes. $\text{path}(u \to v)$ is a simple path connecting nodes u and v. Sometimes to simplify notation we will omit $\text{path}(*)$ and denote it as $u \to v$. $|p|$ denotes the number of nodes in a path p. $\text{parent}(v)$ is the first node on the path from v to the root ($\text{parent}(\text{root}) = \text{null}$). $\text{children}(v) = \{u \mid \text{parent}(v) = u\}$. $\text{dist}(u, v)$ is the number of edges on a simple path connecting u and v. Please note that $\text{dist}(v, u) = \text{dist}(u, v) = |u \to v| - 1$, $\text{depth}(v) = \text{dist}(v, \text{root})$, $\text{subtree}(v) = \{u \mid v \in \text{path}(u \to \text{root})\}$, $\text{height}(v) = \max_{l \in \text{subtree}(v)} \text{dist}(v, l)$, $\text{label}(u \to v)$ is a label of $\text{path}(u \to v)$ constructed as a concatenation of characters labeling its consecutive edges, $\text{label}_d(u \to v) = \text{label}(u \to v)[1..d]$. Figures 2, 3, 4, 5, 6, 7 and 8 show examples of those notations.

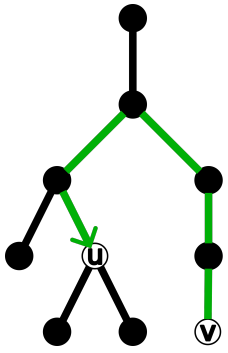


Fig. 2: $\text{path}(v, u)$ consists of 6 nodes and 5 edges. $|\text{path}(v, u)| = 5$

[1] `https://students.mimuw.edu.pl/~lk385775/string_tree_covers_ref_impl.zip`

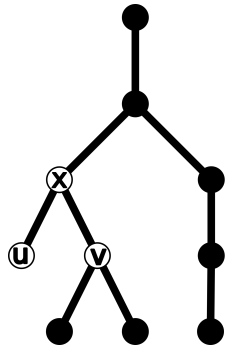


Fig. 3: $\text{children}(x) = \{u, v\}$, $\text{parent}(u) = \text{parent}(v) = x$

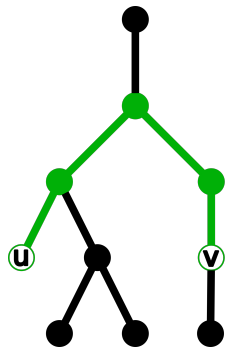


Fig. 4: $|\,\text{path}(u \to v)| = |\,\text{path}(v \to u)| = \text{dist}(u, v) + 1 = 5$

Let us define $\text{childrenHeights}(v) = \sum_{w \in \text{children}(v)} \text{height}(w)$, $\text{maxChildHeight}(v) = \max_{w \in \text{children}(v)} \text{height}(w)$ (0 if $\text{children}(v) = \emptyset$), and $\text{superHeight}(v) = \text{childrenHeights}(v) - \text{maxChildHeight}(v)$.

Lemma 1 *For a rooted tree T with n nodes, we have*

$$\sum_v \text{superHeight}(v) = \sum_v (\text{childrenHeights}(v) - \text{maxChildHeight}(v)) \leq n$$

Proof. The following proof is based on Second Heights lemma proof from [16]. For a node v we define $\text{MaxPath}(v)$ as the longest path from v to a leaf in $\text{subtree}(v)$. ($|\,\text{MaxPath}(v)| = \text{height}(v)$). Initially, we choose (one of possibly many) MaxPath(root), then we remove this path (both nodes and edges) and choose the longest paths for roots of resulting subtrees. We continue in this way and obtain a decomposition of the tree into node-disjoint longest paths.

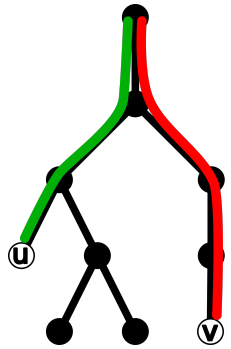


Fig. 5: depth(u) = 3, depth(v) = 4

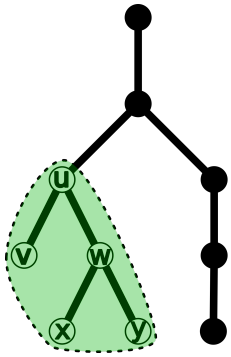


Fig. 6: subtree(u) = $\{u, v, w, x, y\}$

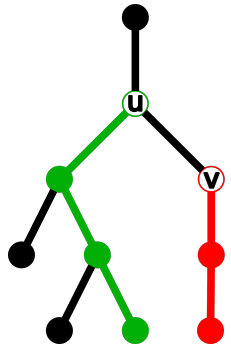


Fig. 7: height(u) = 3, height(v) = 2

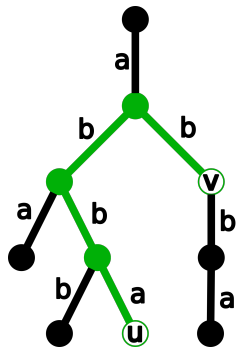


Fig. 8: label$(u \to v) = abbb$, label$(v \to u) = bbba = (abbb)^R$, label$_2(u \to v) = ab$

Let FirstChild(v) denote a child of v which belongs to the same path in the decomposition and OtherChildren$(v) = \{w \in \text{children}(v) : w \neq \text{FirstChild}(v)\}$. We have $\sum_v \text{superHeight}(v) = \sum_{v;w\in\text{OtherChildren}(v)} |\,\text{MaxPath}(w)| \leq n$ since all selected longest paths are node-disjoint. This sum is a sum of the lengths of all removed paths, excluding the one removed in the first step of the algorithm. □

Let us define secondHeight(v) as the height of a second highest child of v (or 0 if $|\,\text{children}(v)| < 2$). We have childrenHeights$(v) - \text{maxChildHeight}(v) \geq \text{secondHeight}(v)$, so

Lemma 2 *(Second height lemma, also used in [16])For a rooted tree T with n nodes, the following inequality holds:*

$$\sum_v \text{secondHeight}(v) \leq n.$$

Pref table is a data structure that is used to store and retrieve information about prefixes of a given string. It is defined as

$$\text{Pref}_S[i] = \max\{d \geq 0 : S[i..(i+d-1)] = S[1..d]\}$$

This data structure can be generalized to rooted trees with character-labeled edges. For a string S, rooted tree T and node $v \in T$, we denote

$$\text{TreePref}_S[v] = \max\{d \geq 0 : \text{label}_d(v \to \text{root}) = S[1..d]\}$$

Lemma 3 *([16] using [17])* TreePref_S *can be computed in $O(n)$ time for a rooted tree T with n nodes over an integer alphabet.*

3 Directed tree cover

Let us consider a directed variant of the string tree cover. For a given rooted tree T, we direct each edge towards the root. For this problem, we will assume

that all edge labels are characters over an integer alphabet. Let us fix some leaf node l. An edge between l and parent(l) can only be covered by a path starting in l and going upwards.

Observation 1 *[16] The cover must be a prefix of* label($l \to$ root).

Let $L = \text{label}(l \to \text{root})$. Let us define $\text{up}(v, k) = \text{parent}(\text{up}(v, k-1))$, $\text{up}(v, 0) = v$. For each $1 \leq d \leq |L|$, we will check if $L[1..d]$ covers all edges of the tree. Let us calculate TreePref_L. Set $\{v : \text{TreePref}_L[v] \geq d\}$ contains nodes (let us denote any of those nodes as v), for which $\text{label}(v \to \text{root})$ matches L on the first d positions. If $\text{label}_d(v \to \text{root}) = L[1..d]$ holds, then then the first d edges on path $v \to \text{root}$ are covered by path $v \to \text{up}(v, d)$.

For each fixed d, we will consider $L[1..d]$. Let us mark all w's, for which $\text{TreePref}_L[w] \geq d$. Edge $w \to \text{parent}(w)$ is covered (by a path having label $= L[1..d]$), if and only if there exists a marked node $u \in \text{subtree}(w)$, having $\text{dist}(w, u) < d$.

We will maintain a data structure to store marked nodes. It will be able to:

- initialize itself with any set of marked nodes (all leaves need to be in this set),
- unmark a marked node,
- query for the largest distance between any node and its closest marked descendant.

3.1 Gaps

All leaves have to be marked at all times - it is the only way to cover leaf edges. For a set M of marked nodes we define $\text{gap}(v) = \min_{u \in M \cap \text{subtree}(v)} \text{dist}(v, u)$ and $\text{MaxGap} = \max_{v \in nodes \setminus \{\text{root}\}} \text{gap}(v)$.

In a data structure, nodes will keep (either directly or indirectly) their current gap value. Let v be any marked node, which is not a leaf. Let $M' = M \setminus \{v\}$ be the set of marked nodes after unmarking v. Let us define $\text{gap}'(w) = \min_{u \in M' \cap \text{subtree}(w)} \text{dist}(w, u)$ – it is a gap function after unmarking v.

Observation 2 *If* $\text{gap}(w) \neq \text{gap}'(w)$*, then* $w \in \text{path}(v \to \text{root})$.

Proof. A gap could change only for those nodes w, for which $v \in \text{subtree}(w)$. □

Lemma 4 *Let* $u, w \in \text{path}(v \to \text{root})$*, and* $\text{depth}(u) > \text{depth}(w)$*. If* $\text{gap}'(u) = \text{gap}(u)$*, then* $\text{gap}'(w) = \text{gap}(w)$.

Proof. If $\text{dist}(w, v) \neq \text{gap}(w)$ then there exists $x \in \text{subtree}(w) \cap M$ $(x \neq v)$, for which $\text{dist}(w, x) = \text{gap}(w)$. Since $x \in M' = M \setminus \{v\}$, then $\text{gap}'(w) = \text{dist}(w, x) = \text{gap}(w)$.

Otherwise, if $\text{dist}(w, v) = \text{gap}(w)$, then $\text{dist}(u, v) = \text{gap}(u)$. Since $\text{gap}'(u) = \text{gap}(u)$, then there exists $x \in \text{subtree}(u) \cap M'$, for which $\text{dist}(u, x) = \text{dist}(u, v) =$

$\text{gap}(u)$. $\text{dist}(w,x) = \text{dist}(w,u) + \text{dist}(u,x) = \text{dist}(w,u) + \text{dist}(u,v) = \text{dist}(w,v)$. And since $x \in \text{subtree}(u) \cap M' \subseteq \text{subtree}(w) \cap M'$, then $\text{gap}'(w) = \text{dist}(w,x) = \text{dist}(w,v) = \text{gap}(w)$. □

Corollary 1 *After unmarking v, we only need to update* gap *for some prefix of nodes on* $\text{path}(v \to \text{root})$.

3.2 Binarisation and path compaction

For the data structure representation, we apply two transformations on tree T:

1. **binarisation** - we will insert artificially created nodes, so that every node, except for the root, has at most 2 children. Let us denote the resulting tree as T'. The number of inserted nodes is $\sum_{v \neq \text{root}} \min(0, \text{degree}(v) - 3) \leq n$, so $|T'| \leq 2n = O(n)$.
2. **path compaction** - we will replace each non-branching path of T' by a single entity, called **compacted path**. Here by non-branching path, we refer to a group of nodes, forming a simple path, each of them having only one child.

The result of those transformations will be called a pseudotree.

The defined gap function will be invariant to those transformations. We will calculate and update its values as if all nodes were located in the original input tree T. We will not take into account gap values calculated for nodes added during binarisation.

The resulting pseudotree will consist of four types of nodes: root, leaves, binary nodes, and implicit nodes. Each non-branching vertical path will be replaced by a compacted path. The remaining edges will be replaced by trivial (that is not having any implicit nodes inside) compacted paths.

Each binary node holds two compacted paths going down and one going up. The root holds some number (possibly one) of compacted paths going down. Each leaf holds only one compacted path going up. Implicit nodes are connected together and contained inside compacted paths. Compacted paths will hold both: nodes they are connected to, and a collection of implicit nodes inside of it.

3.3 Updates

The key observation here is that we do not need to explicitly keep and update the gap sizes of implicit nodes.

`unmark(v)` is an update entry point of the data structure, please refer to Listing 1 for its pseudocode.

On Listing 2 we attach the pseudocode of `walkAndUpdate` function, which is called by `unmark` to propagate new gap values upwards.

Let us denote any implicit node v, and its compacted path as p. When we `unmark` v:

– If v is the only marked node on p, then the new gap value for p.top is p.length+p.nodeDown.gap and this value needs to be passed to `walkAndUpdate` to propagate it upwards.

Listing 1: Unmark function pseudocode.

```
function unmark (v)
    if isBinary(v) then
        v.isMarked = false
        walkAndUpdate(v)
    else if isImplicit(v) then
        markedUp = v.markedUp
        markedDown = v.markedDown
        v.markedUp = v.markedDown = null
        if markedUp ≠ null then
            markedUp.markedDown = markedDown
        if markedDown ≠ null then
            markedDown.markedUp = markedUp
        path = v.path
        if path.lowestMarked = v then
            path.lowestMarked = markedUp
            if markedUp ≠ null then
                gap = path.nodeDown.gap + dist(markedUp, path.bottom)
                maxGap = max(maxGap, gap)
        if path.highestMarked = v then
            path.highestMarked = markedDown
            if path.highestMarked ≠ null then
                gap = dist(path.highestMarked, path.top)
            else
                gap = path.length + path.nodeDown.gap
            path.topGap = gap
            maxGap = max(maxGap, gap)
            walkAndUpdate(path.nodeUp)
        if markedUp ≠ null and markedDown ≠ null then
            maxGap = max(maxGap, dist(markedUp, markedDown) - 1)
```

Listing 2: Auxilary functions needed for gap values updates.

```
global maxGap = 1
function calcGapBinaryNode (v)
    if v.isMarked then
        return 0
    else
        return min(v.pathLeft.topGap, v.pathRight.topGap)+!v.isFake;
function walkAndUpdate (v)
    while (not isRoot(v)) and v.gap < calcGapBinaryNode(v) do
        v.gap = calcGapBinaryNode(v)
        path = v.pathUp
        if path.lowestMarked ≠ null then
            gap = v.gap + dist(path.bottom, path.lowestMarked) + 1
            maxGap = max(maxGap, gap)
            break
        else
            path.topGap = v.gap + dist(path.top, path.bottom) + 1
            maxGap = max(maxGap, path.topGap)
            v = path.nodeUp
```

- Otherwise, if v is the highest marked node on p, then we can calculate the new longest gap as a distance between $p.top$ and the new highest marked node on p. Then it needs to be propagated upwards using `walkAndUpdate`.
- Otherwise, if v is the lowest marked node on p, then the longest new gap is equal to p.nodeDown.gap plus the distance between p.bottom and child of the new lowest marked node on p. This value does not get propagated upwards, since there exist marked nodes on p, other than v.
- Otherwise, v is located between two other marked implicit nodes. We can calculate the longest new gap using the distance between them.

Similarly, whenever `walkAndUpdate` tries to traverse a compacted path p (to proceed from a binary node on its lower end to the one on its upper end), it checks if there exists any marked node on this path. It can use the gap calculated for the binary node connected to the lower end of that path, in order to calculate the new gap size for implicit nodes of p. It needs this value to update `maxGap`.

Listing 3 contains the final high-level algorithm for computing all directed covers of a given rooted tree.

Lemma 5 *The total amortized time cost of maintaining the gap data structure is $O(n)$.*

Proof. `unmark` itself does only $O(1)$ amount of work. `walkAndUpdate` runs in time proportional to the number of touched binary nodes. By *touched nodes*, we refer to those nodes, for which node.gap has changed. Since node.gap can only increase, the total amount of time used by `walkAndUpdate` is limited by $\sum_{v \in \text{binary nodes}} \text{FinalGap}(v)$. (By FinalGap($v$) we refer to v.gap after termination of the algorithm.)

Listing 3: Computing all directed covers of a given rooted tree.

```
Fix any leaf l. Let us denote L = label(l → root)
Calculate TreePref_L - this can be done in O(n) as we are working with integer
  alphabet
Initialize gap data structure - apply binarization and path compression
Set all real nodes (that is those which were not created during binarisation) as
  marked
for k := 0 to min_{v∈leaves} TreePref_L[v] − 1 do
    for v ∈ nodes and TreePref_L[v] = k do
        unmark(v)
    if maxGap < k then
        report that L[1..k] is a cover
```

All leaves are always marked, so $\text{FinalGap}(v) \leq \min_{l \in \text{leaves} \cap \text{subtree}(v)} \text{dist}(v, l)$ (here we refer to leaves, subtree, and dist in regards to T'). Each binary node v has two children, so $\text{FinalGap}(v) \leq \text{secondHeight}(v)$. Second heights lemma (Lemma 2) applied to T' implies that $\sum_v \text{secondHeight}(v) \leq 2n$. This indicates that the total number of binary node gap updates done by `walkAndUpdate` is bounded by $2n$. This also proves the complexity of the complete directed cover computing algorithm. □

Theorem 1. *All covers of a directed rooted tree labeled with characters over an integer alphabet can be computed in $O(n)$ time and $O(n)$-space.*

The only step of the algorithm, which requires characters labeling edges to be over an integer alphabet, is the subroutine computing TreePref_L 3 from [17]. Thus

Corollary 2 *Covers of a directed tree labeled with characters over a general alphabet can be computed in $O(n)$ time and $O(n)$-space if TreePref_L, for $L =$ label of some path from a leaf to the root, is given.*

Comparison with the algorithm from [16]. Instead of maintaining gap data structure over a transformed tree – they maintain the so-called chain decomposition of an input tree, where each marked node is an end of some chain, and the chain from any unmarked node leads to the closest marked node in its subtree – our maxGap is equal to the length of the longest chain in their data structure. Chain description is stored in the top node of each chain. None of the other chain nodes store any information about the chain they currently belong to. A data structure from [1] is used to query for a top node of each chain. The time complexity of such a query is $O(\log n / \log \log n)$. This impacts the overall time complexity of their algorithm, which is $O(n \log n / \log \log n)$.

4 Undirected tree cover

The cover of an undirected tree is quite a different problem than that of a directed tree. String S is a cover of an undirected tree T, if we can pick a set of simple

paths M, each of them having a label equal to S, such that for each edge from the tree, there should exist at least one path from M, going through that edge.

Let us root the input tree in any node.

Observation 3 *For any leaf l, edge $l \to \text{parent}(l)$ can only be covered by a path starting or finishing in l.*

We will fix some leaf l. Let us denote set of paths that can cover edge $l \to \text{parent}(l)$ as $P = \{l \to v \mid v \in T \setminus \{l\}\} \cup \{v \to l \mid v \in T \setminus \{l\}\}$. We have $|P| = 2n - 2$.

The set of candidates for a cover is naturally induced by P. Let us denote it by $C = \{\text{label}(p) \mid p \in P\}$. Thus $|C| \leq 2n - 2 = O(n)$.

Corollary 3 ([16]) *The set of candidates has $O(n)$ elements.*

4.1 Match tables

Let us fix a candidate S. For each node v we will consider those paths, for which v is the highest point (v is the highest point of p if it has the smallest depth among all its nodes). We will try to match a prefix of S, coming from a subtree of some $u \in \text{children}(v)$, with a prefix of S^R coming from another subtree. A prefix of S concatenated with the reverse of a prefix of S^R of proper length makes S. Found matches-paths will be saved on the side.

We will be going from the bottom to the top of the tree (the top being root, and the bottom being leaves). For each node v we will compute two *match tables*: dynamic arrays of linked lists: A and B. They have the following properties:

- $|A| = |B| = \text{height}(v)$
- $A[i]$ and $B[i]$ contain nodes from subtree(v).
- For each $u \in A[i]$ (and for each $u \in B[i]$), $\text{dist}(u, v) = i$.
- If there exists node u having $\text{label}(u \to v) = S[1..i]$, then $A[i]$ is not empty and contains a node y such that $\text{label}(y \to v) = S[1..i]$. Otherwise, it might be empty or might contain some nodes with label different than $S[1..i]$.
- If there exists node u having $\text{label}(u \to v) = S^R[1..i]$, then $B[i]$ is not empty and contains a node y such that $\text{label}(y \to v) = S^R[1..i]$. Otherwise, it might be empty or might contain some nodes having label different than $S^R[1..i]$.
- For all nodes $u \in A[i]$, $\text{label}(u \to v)$ is the same.
- For all nodes $u \in B[i]$, $\text{label}(u \to v)$ is the same.

Each table is held by a data structure, which allows amortized $O(1)$-time insertions to the front and $O(1)$-time random access. Such data structure can be implemented similarly to `std::vector` from C++ STL [18] or Dynamic Table from [6]: by keeping a pointer to a chunk of allocating memory, and lazily moving its content to a chunk twice as big when space runs out. The only difference is that we will push to the front, not to the back, and use the current size to calculate the offset for $O(1)$-time random access. We will call it `FrontVector`.

If v is a leaf, then match tables for v are trivial: A.size() = B.size() = 1, and $A[0] = B[0] = \{v\}$. (By V.size() we denote the number of elements of a table V. For the match table represented by `FrontVector` it is the difference between its capacity and offset.) Otherwise, we need to calculate A and B for v. At first we "claim" match tables calculated for the highest child. That is from the child, whose subtree is the highest among all v's children, in case of ambiguity, whichever child can be picked. Match tables calculated for a node will be used only by its parent, so we can immediately claim its ownership.

Then we need to push $\{v\}$ to the front of A and B, iterate over the remaining children, and join its match tables. We will be also looking for matches during that process.

At the beginning of the algorithm, we precalculate TreePref_S and TreePref_{S^R}, which per Lemma 3 can be done in $O(n)$ [17].

This lemma requires the labeling alphabet to be integer. If this is not the case, then we can convert it to an equivalent integer alphabet in $O(n^2)$ time and $O(n)$-space. Equivalent in the sense, that equivalence relation on edges, based on equality of their labeling characters, will look exactly the same. Since we represent a cover as an ordered pair of endpoints of a path having its label as an actual cover, the output of our algorithms will not change after this transformation. Now we are able for a given v and $u \in \text{subtree}(v)$ check if $\text{label}(u \to v)$ is a prefix of S (or S^R). This condition is equivalent to checking if $\text{TreePref}_S[u] \geq \text{dist}(u, v)$ ($\text{TreePref}_{S^R}[u] \geq \text{dist}(u, v)$).

Listing 4: auxiliary procedures used to find matches in match tables and mark found matches.

```
function matchAndMark (v, A, B, i, j)
    // i + j == len(W)
    clearA(v, A, i)
    clearB(v, B, j)
    if !A[i].empty() and !B[j].empty() then
        for u ∈ (A[i] ∪ B[j]) do
            markVerticalPath(u, v) // markVerticalPath works in O(1) and
             will be described later
        clearButOne(A, i)
        clearButOne(B, j)
function findMatches (v, A, B, A', B', height)
    lowerBound = max(len(S) - A.size() + 1, 0)
    upperBound = min(height, len(S)) + 1
    for i := lowerBound to upperBound do
        matchAndMark(v, A, B', len(S) - i, i)
        matchAndMark(v, A', B, i, len(S) - i)
```

Let us denote current tables as A and B, and match tables calculated for some other child as A' and B'. Let $h = A'$.size() = B'.size() = height of the subtree rooted in that child. For $1 \leq i \leq h$, we consider matches between $A'[i]$ and

$B[|S|-i]$. To do that we first need to check if $A'[i]$ contains valid candidates, that is nodes u, for which label$(u \to v) = S[1..i]$. Please recall that every $u \in A'[i]$ has the same label$(u \to v)$, so it is sufficient to check that condition for any element of $A'[i]$ - we will use the first one. In our pseudocode, we will use `clearA` (and `clearB` for match table B) procedures to refer to this step. Please refer to Listing for its pseudocode.

Listing 5: Procedures responsible for removing invalid or redundant nodes from match table entries.

```
function clearA (v, A, i)
    if not A[i].empty() then
        // O(1) by precalculated TreePref_S
        if label(A[i].front() → v) ≠ S[1..i] then
            A[i].clear() // O(|A[i]|)
function clearB (v, B, i)
    if not B[i].empty() then
        // O(1) by precalculated TreePref_SR
        if label(B[i].front() → v) ≠ S^R[1..i] then
            B[i].clear() // O(|B[i]|)
function clearButOne (matchTable, i)
    while matchTable[i].size() > 1 do
        matchTable[i].pop_back()
```

Similar procedure must be performed for $B[|S|-i]$ - it should contain nodes u having label$(u \to v) = S^R[1..(|S|-i)]$.

After that step, for each $u_1 \in A'[i]$ and $u_2 \in B[1..(|S|-i)]$ we have label$(u_1 \to u_2) = S$, so we can mark both $u_1 \to v$ and $u_2 \to v$ as covered. If both $A'[i]$ and $B[1..(|S|-i)]$ are not empty, then we will delete all elements of $A'[i]$ except for one and $B[1..(|S|-i)]$ except for one. This a crucial step to the complexity of the algorithm. Here we will only present simplified intuition why we can do this, proof of correctness will be discussed later.

Imagine that we would not delete any elements of $A'[i]$. When seeking matches for some v' - ancestor of v, in terms of $A'[i]$ we will care about two things:

- $A'[i]$ (which will become $A[i + \text{dist}(v, v')]$) is not empty - then we can find match it with some $B[j]$ coming from some other child of v' and, as a consequence mark all paths $v' \to u$ for $u \in B[j]$.
- if we indeed find a match, all paths $v' \to u$ for $u \in A'[i]$ will become marked.

Please note that since each path $v \to u$ is already marked, if we mark $v' \to v$ then all paths $v' \to u$ will become marked, so marking any path $v' \to u$ marks all of them.

Both of those objectives will still be satisfied if we delete all but one element of $A'[i]$. Similar argument can be conducted for $B[i]$, $A[i]$ and $B'[i]$. In pseudocodes, we will use `clearButOne` subroutine to refer to this step, please refer to the 5 its pseudocode.

Then we follow the same procedure to find matches between $B'[i]$ and $A[|S|-i]$.

After we have considered all valid matches, we can merge A' into A, and B' into B. For $1 \leq i \leq h$, we have to check if $A[i]$ contains nodes u, having label$(u \to v) = S[1..i]$. To do that, it is enough to check that condition for $u =$ first element of $A[i]$ (all nodes in $A[i]$ have the same path to the root). If not, we can clear $A[i]$. We apply the exact same procedure to $A'[i]$. After that, both $A'[i]$ and $A[i]$ contain only valid candidates, so we can move all nodes from $A'[i]$ into $A[i]$ (in $O(1)$-time).

To merge B' into B we execute the same algorithm, but instead of comparing labels of paths to root with S, we compare them with S^R.

4.2 Complexity and correctness

Let us denote `calcAB(root)` as an entry point for the entire procedure. It calls recursively itself, resulting in a call to `calcAB(v)` once for every v of the tree ($O(n)$ calls). It also iterates over all children ($O(|\text{edges}|) = O(n)$ in total). A single call consumes amortized constant time for inserting $\{v\}$ to the front of match tables - implemented as a `push_front` on an instance of `FrontVector`. So $O(|\text{edges}|) + O(|\text{nodes}|) = O(n)$ of total time consumed.

For each other child (let us denote its height as h) it calls: `findMatches` once, and `clearA`, `clearB` h times - to clear each entry of match tables propagated from that child. `findMatches` calls `matchAndMark` h times. If we disregard time spent on `clearA`, `clearB`, `clearButOne` and `matchAndMark`, then total time spent in `findMatches` and `calcAB` is bounded by $O(\sum_v \text{superHeight}(v)) + O(n) = O(n)$ (by Sum of heights, Lemma 1).

`clearA`, `clearB`, and `clearButOne` run in time proportional to the number of deleted nodes. Each node is inserted only twice - each node v is inserted only in `calcAB(v)`. When a node is deleted, it is deleted permanently. Match tables are never copied - only moved and merged. It implies that the total number of deletions is bounded by the total number of insertions $= 2n$, so the time cost of clearing functions is bounded by $O(n)$. Please refer to the 5 for pseudocodes of those functions.

`matchAndMark` calls: `clearA`, `clearB`, and `clearButOne`. If we disregard that, it runs in time proportional to the number of deleted nodes by auxiliary subroutines: loop for $u \in (A[i] \cup B[j])$ iterates over $|A[i]| + |B[j]|$ elements. Calls to `clearButOne(A, i)` delete $|A[i]| - 1$ elements, calls to `clearButOne(B, j)` delete $|B[j]| - 1$ elements. Thus the total cost of `matchAndMark` is also bounded by $O(n)$.

Corollary 4 *We can compute match tables for all nodes of a tree using $O(n)$ time.*

Finally, we verify if the found set of marked paths indeed covers the whole tree. This procedure comes from [16], we repeat it here for completeness. We will store a counter for each node. All marked paths are vertical - one of their ends is a descendant of the other. For each marked path we will add 1 to the counter of the lower, and add -1 to the counter of the higher end - this is what

`markVerticalPath` from pseudocode serves for. Let us fix any node v. The sum of counters in the subtree rooted in v is equal to the number of marked paths going through the edge from v to its parent. With a single DFS from the root we can calculate sums for every subtree. All edges of a tree are covered if and only if $\sum_{u \in \text{subtree}(v)} \text{counter}(u)$ is positive for every node v except for the root.

Now all that is left, is the proof of correctness.

Lemma 6 `matchAndMark` *marks only simple paths* p *having* $\text{label}(p) = S$.

Proof. Merge function maintains invariant, that for any i, every $u \in A[i]$ ($B[i]$) has the same $\text{label}(u \to \text{root})$. This implies that after calling `clearA(v, A, i)` (`clearB(v, B, i)`), for all nodes $u \in A[i]$ ($B[i]$), $\text{label}(u \to v)$ is a prefix of S (S^R for $u \in B[i]$). Thus in `matchAndMark`, after clearing A and B, for any $u \in A[i]$ and any $x \in B[j]$, we have $\text{label}(u \to x) = S$, and all marked paths have label S. And since `matchAndMark` matches nodes coming from different subtrees, then all marked paths are simple. □

Lemma 7 *If* `matchAndMark` *did not delete nodes using* `clearButOne` *subroutine, then we would mark every path* p *in the tree having* $\text{label}(p) = S$.

Proof. Let us fix any path p: $x \to y$, $\text{label}(x \to y) = S$. Let $v = \text{LCA}(x, y)$ (LCA is the lowest common ancestor of two nodes) be the lowest node on that path (the closest to the root).

Let us assume that $x, y \neq v$. Let $i = \text{dist}(x, v)$ and $j = \text{dist}(y, v)$ ($i + j = |S|$). Since $\text{label}_i(x \to \text{root}) = S[1..i]$, then $x \in A[i]$ calculated for v. Since $\text{label}(y \to \text{root})[1..j] = S^R[1..j]$ then $y \in B[j]$. Since x and y come from subtrees of different children of v, then at some point of `calcAB` we will have $x \in A[i-1]$ and $y \in B'[j-1]$ (or $x \in A'[i-1]$ and $y \in B[j-1]$). And at that point `matchAndMark`, called by `findMatches`, would mark path $x \to y$.

If $x = v$ then $\text{label}(y \to v) = S^R$, $y \in B[|S|]$, and $x \to y$ will be marked by function `matchAndMark(v, A, B, 0, |S|)`, called by `calcAB(v)`.

If $y = v$ then $\text{label}(x \to v) = S$, $x \in A[|S|]$, and $x \to y$ will be marked by function `matchAndMark(v, A, B, |S|, 0)` called by `calcAB(v)`. □

The next lemma establishes that deleting some nodes in `matchAndMark` and leaving only one representative, even if labels of their paths to v are still valid prefixes of S/S^R, is indeed a correct action.

Lemma 8 *If* $\text{label}(a \to v) = \text{label}(b \to v)$ *and both* $a \to v$ *and* $b \to v$ *are marked, then we can skip propagating either one upwards.*

Proof. We have $\text{depth}(a) = \text{depth}(b)$, and for any v' being ancestor of v, we have $\text{label}(a \to v') = \text{label}(b \to v')$. Thus we cannot tell apart a and b after propagating them upwards - when we later look for matches only labels matter. Moreover if at some point later we will decide to mark $a \to v'$, then $b \to v'$ will become fully marked as well. This is because $b \to v'$ can be decomposed into two edge-disjoint paths: $b \to v$ and $v \to v'$. Similarly $a \to v'$ can be decomposed into edge-disjoint $a \to v$ and $v \to v'$. This means that $v \to v'$ becomes marked when $a \to v$ is marked. All edges on path $b \to v$ are already marked. □

Those three lemmas combined together prove that `matchAndMark` will mark all, and only edges covered by S. Thus they are completing a proof of the main theorem of this section.

Theorem 2. *We can compute all undirected covers in $O(n^2)$ time and $O(n)$-space.*

The general idea of the presented algorithm is similar to the centroid decomposition algorithm from [16]. We rely on the same set of candidates, and in the same way, we check if the set of marked paths covers all edges of the tree. However, calculating match tables A and B to mark all covered edges is a completely new idea.

Achieved $O(n^2)$ time $O(n)$-space complexity improves results from [16] ($O(n^2)$ time and space, or $O(n^2 \log n)$ time $O(n)$-space), but is still superlinear. Further work will be focused on research on $o(n^2)$ time algorithm or finding a conditional lower bound.

References

1. Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA*, pages 534–544. IEEE Computer Society, 1998.
2. Alberto Apostolico and Andrzej Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theor. Comput. Sci.*, 119(2):247–265, 1993.
3. Alberto Apostolico, Martin Farach, and Costas S. Iliopoulos. Optimal superprimitivity testing for strings. *Inf. Process. Lett.*, 39(1):17–20, 1991.
4. Dany Breslauer. An on-line string superprimitivity test. *Inf. Process. Lett.*, 44(6):345–347, 1992.
5. Srecko Brlek, Nadia Lafrenière, and Xavier Provençal. Palindromic complexity of trees. In Igor Potapov, editor, *Developments in Language Theory - 19th International Conference, DLT 2015, Liverpool, UK, July 27-30, 2015, Proceedings*, volume 9168 of *Lecture Notes in Computer Science*, pages 155–166. Springer, 2015.
6. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.
7. Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, Wojciech Tyczyński, and Tomasz Waleń. The maximum number of squares in a tree. In Juha Kärkkäinen and Jens Stoye, editors, *Combinatorial Pattern Matching - 23rd Annual Symposium, CPM 2012, Helsinki, Finland, July 3-5, 2012. Proceedings*, volume 7354 of *Lecture Notes in Computer Science*, pages 27–40. Springer, 2012.
8. Patryk Czajka and Jakub Radoszewski. Experimental evaluation of algorithms for computing quasiperiods. *Theor. Comput. Sci.*, 854:17–29, 2021.
9. Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing maximal palindromes and distinct palindromes in a trie. In Jan Holub and Jan Zdárek, editors, *Prague Stringology Conference 2019, Prague, Czech Republic, August 26-28, 2019*, pages 3–15. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2019.

10. Pawel Gawrychowski, Tomasz Kociumaka, Wojciech Rytter, and Tomasz Waleń. Tight bound for the number of distinct palindromes in a tree. In Costas S. Iliopoulos, Simon J. Puglisi, and Emine Yilmaz, editors, *String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings*, volume 9309 of *Lecture Notes in Computer Science*, pages 270–276. Springer, 2015.
11. Pawel Gawrychowski, Tomasz Kociumaka, Wojciech Rytter, and Tomasz Waleń. Tight bound for the number of distinct palindromes in a tree. *The Electronic Journal of Combinatorics*, 30, 04 2023.
12. Tomasz Kociumaka, Jakub Pachocki, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Efficient counting of square substrings in a tree. *Theor. Comput. Sci.*, 544:60–73, 2014.
13. Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. String powers in trees. *Algorithmica*, 79(3):814–834, 2017.
14. Neerja Mhaskar and William F. Smyth. String covering: A survey. *CoRR*, abs/2211.11856, 2022.
15. Dennis W. G. Moore and William F. Smyth. A correction to "An optimal algorithm to compute all the covers of a string". *Inf. Process. Lett.*, 54(2):101–103, 1995.
16. Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, and Wiktor Zuba. String covers of a tree. In Thierry Lecroq and Hélène Touzet, editors, *String Processing and Information Retrieval - 28th International Symposium, SPIRE 2021, Lille, France, October 4-6, 2021, Proceedings*, volume 12944 of *Lecture Notes in Computer Science*, pages 68–82. Springer, 2021.
17. Tetsuo Shibuya. Constructing the suffix tree of a tree with a large alphabet. In *Algorithms and Computation*, pages 225–236, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
18. Bjarne Stroustrup. *The C++ programming language - special edition (3. ed.)*. Addison-Wesley, 2007.
19. Ryo Sugahara, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Efficiently computing runs on a trie. *Theor. Comput. Sci.*, 887:143–151, 2021.