An optimal algorithm for computing all subtree repeats in trees

Tomáš Flouri[1,*], Kassian Kobert[1,**], Solon P. Pissis[1,2,***], and Alexandros Stamatakis[1,3]

[1] Heidelberg Institute for Theoretical Studies, Germany
[2] Florida Museum of Natural History, University of Florida, USA
[3] Karlsruhe Institute of Technology, Institute for Theoretical Informatics, Postfach 6980, 76128 Karlsruhe

Abstract. Given a labeled tree T, our goal is to group repeating subtrees of T into equivalence classes with respect to their topologies and the node labels. We present an explicit, simple, and time-optimal algorithm for solving this problem for unrooted unordered labeled trees, and show that the running time of our method is linear with respect to the size of T. By unordered, we mean that the order of the adjacent nodes (children/neighbors) of any node of T is irrelevant. An unrooted tree T does not have a node that is designated as root and can also be referred to as an undirected tree. We show how the presented algorithm can easily be modified to operate on trees that do not satisfy some or any of the aforementioned assumptions on the tree structure; for instance, how it can be applied to rooted, ordered or unlabeled trees.

1   Introduction

Tree data structures are among the most common and well-studied of all combinatorial structures. Tree structures are present in a wide range of applications, such as, in the implementation of functional programming languages [12], term-rewriting systems [11], programming environments [2], code optimization in compiler design [1], code selection [8], theorem proving [13], and computational biology [14].

Thus, efficiently extracting the repeating patterns in a tree structure, represents an important computational problem. Recently, Christou *et al.* [5] presented a linear-time algorithm for computing all subtree repeats in *rooted ordered unlabeled* trees. In [4], Christou *et al.* extended this algorithm to compute all subtree repeats in *rooted ordered labeled* trees in linear time *and* space.

The limitation of the aforementioned results is that they cannot be applied to *unrooted* or *unordered* trees. By unrooted, we mean that the input tree does not have a dedicated root node; and, by unordered, we mean that the order of

the descendant nodes (children) of any node of the tree is irrelevant. Such trees are a generalization of rooted ordered trees, and, hence, they arise naturally in a broader range of real-life applications. For instance, unrooted unordered trees are used in the field of (molecular) phylogenetics [7, 16].

The field of molecular phylogenetics deals with inferring the evolutionary relationships among species using molecular sequencing technologies and statistical methods. Phylogenetic inference methods typically return unrooted unordered labeled trees that represent the evolutionary history of the organisms under study. These trees depict evolutionary relationships among the molecular sequences of extant organisms (living organisms) that are located at the tips (leaves) of those trees and hypothetical common ancestors at the inner nodes of the tree. With the advent of so-called next-generation sequencing technologies, large-scale multi-national sequencing projects such as, for instance, 1KITE[4] (1000 insect transcriptome sequencing project) emerge. In these projects, large phylogenies that comprise thousands of species and massive amounts of whole-transcriptome or even whole-genome molecular data need to be reconstructed.

Provided a fixed multiple sequence alignment (MSA) of the sequences—representing species—under study, the goal of phylogenetic inference is to find *the* tree topology that best explains the underlying data, using a biologically reasonable optimality criterion—a scoring function for the trees. One such optimality criterion is *Maximum Likelihood* (ML) [6]. Finding the optimal tree under ML is known to be NP-complete [3]. Note that the number of possible unrooted tree topologies for n species, located at the tips, grows super-exponentially with n. Therefore, widely-used tools for ML-based inference of phylogenies, such as RAxML [15] and PHYML [9], rely on heuristic search strategies for exploring the immense tree space.

The likelihood of each candidate tree topology T is calculated by computing the conditional likelihoods at each inner node of T. The conditional likelihoods are computed independently for each *site* (column in the MSA). The conditional likelihoods are computed via a post-order traversal of T starting from a virtual root. Note that, as long as the statistical model of evolution is time-reversible (i.e. evolution occurred in the same way if followed forward or backward in time) the likelihood score is invariant with respect to where in T the virtual root has been placed.

In phylogenetic inference software, a common technique for optimizing the likelihood function, which typically consumes $\approx 95\%$ of total execution time, is to eliminate duplicate *sites* (equivalent columns in the MSA). This is achieved by compressing identical sites into site patterns and assigning them a corresponding weight. This can be done because duplicate sites yield exactly the same likelihood *iff* they evolve under the same statistical model of evolution. When two sites are identical, this means that the leaves of the tree are labeled equally. Consider a forest of trees with the same topology, where, for each tree, the labels are defined by the molecular data stored at a particular site of the MSA and the position of the tips. Knowing equivalent subtrees within such a forest would allow someone

---

[4] http://www.1kite.org/

to minimize the number of operations required to compute the likelihood of a phylogenetic tree. This can be seen as a generalization of the site compression technique.

Our contribution. In this article, we extend the series of results presented in [5] and [4] by introducing an algorithm that computes all subtree repeats in *unrooted unordered labeled* trees in linear time and space. The importance of our contribution is underlined by the fact that the presented algorithm can be easily modified to work on trees that do not satisfy some or any of the above assumptions on the tree structure; e.g. it can be applied to rooted, ordered or unlabeled trees.

2 Preliminaries

2.1 Basic definitions

An unrooted unordered tree is an undirected unordered acyclic connected graph $T = (V, E)$ where V is the set of nodes and E the set of edges such that $E \subset V \times V$ with $|E| = |V| - 1$. The number of nodes of a tree T is denoted by $|T| := |V|$. An *alphabet* Σ is a finite, non-empty set whose elements are called *symbols*. A *string* over an alphabet Σ is a finite, possibly empty, string of symbols of Σ. The length of a string x is denoted by $|x|$, and the concatenation of two strings x and y by xy. A tree T is *labeled* if every node of T is labeled by a symbol from some alphabet Σ. Different nodes may have the same label.

A *tree center* of an unrooted tree $T = (V, E)$ is the set of all vertices such that the greatest node distance to any leaf is minimal. An unrooted tree T has either one node that is a tree center, in which case it is called a *central tree*, or two adjacent nodes that are tree centers, in which case it is called a *bicentral tree* [10]. Let $\hat{T}(T) = (\hat{V}, \hat{A})$ be the rooted tree on $\hat{V} = V \cup \{r\}$, where $\hat{A}$ is defined such that $|\hat{A}|$ is minimal with $(u, v) \in \hat{A}$ only if $\{u, v\} \in E$ and each node other than r is reachable from one central point. If T is a bicentral tree, we add the additional root node r to V and add two edges to $\hat{A}$, namely (r, v) and (r, u), where v and u are the central points of T. Otherwise, if T is a central tree, with tree center u, we set $r := u$ and thus $\hat{V} = V$.

We call $u \in V$ a *child* of v iff $(v, u) \in \hat{A}$. In this case, we call v the *parent* of u and define $parent(u) := v$. We call u and u' *siblings iff* there exists a node $v \in \hat{V}$ such that $(v, u), (v, u') \in \hat{A}$. Note that under this definition two central points of a bicentral tree are siblings of each other.

The (rooted) subtree that is obtained by removing edge $\{v, u\}$ and contains node v as its root node is denoted by $\hat{T}(v, u)$. We consider only *full subtrees*, i.e. subtrees which contain all nodes and edges that can be reached from v when only the edge $\{v, u\}$ is removed from the tree. The special case $\hat{T}(v, v)$ denotes the tree containing all nodes that is rooted in v. For simplicity, we refer to $\hat{T}(v, parent(v))$ as $\hat{T}(v)$. The *diameter* of an unrooted tree T is denoted by $d(T)$ and is defined as the number of edges of the longest path between any two *leafs* (nodes with degree 0) of T. The *height of a rooted (sub)tree* $\hat{T}(v, u)$ of some tree
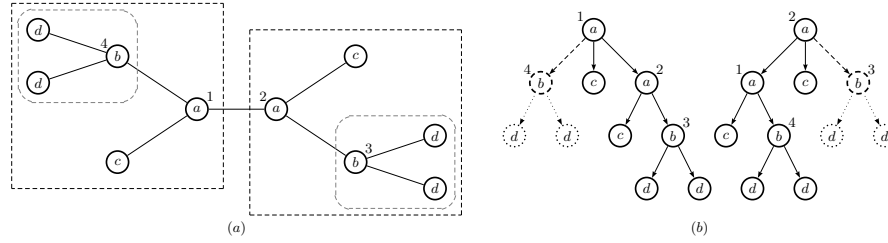
Fig. 1. (a) An *unrooted tree T* consisting of 10 nodes; a *non-overlapping subtree repeat* $R = \{(3, 2), (4, 1)\}$ is marked with dashed rounded rectangles; another *non-overlapping subtree repeat* containing the trees $\hat{T}(1, 2), \hat{T}(2, 1)$ is marked with dashed rectangles (b) An *overlapping subtree repeat* $R = \{(2, 3), (1, 4)\}$ of T resulting from the deletion of the dashed edge and its corresponding dotted subtree. This is an overlapping subtree repeat since nodes 1 and 2–and the node labeled by c–are contained in both subject trees. A *total repeat* $R = \{(1, 1), (2, 2)\}$ of T can be obtained by keeping all the edges and rooting T in node 1 ($\hat{T}(1, 1)$) and 2 ($\hat{T}(2, 2)$), respectively

T, denoted by $h(v, u)$, is defined as the number of edges on the longest path from the root v to some leaf of $\hat{T}(v, u)$. The *height of a node v*, denoted by $h(v)$, is defined as the length of the longest path from v to some leaf in $\hat{T}(T)$.

For simplicity, in the rest of the text, we denote: a rooted unordered labeled tree by $\hat{T}$; an unrooted unordered labeled tree by T; and the rooted (directed) version of T by $\hat{T}(T)$, as defined above.

2.2 Subtree repeats

Two trees $\hat{T}_1 = (V_1, A_1)$ and $\hat{T}_2 = (V_2, A_2)$ are *equal* ($\hat{T}_1 = \hat{T}_2$) if there exists a bijective mapping $f : V_1 \rightarrow V_2$ such that the following two properties hold

$$(v_1, v_2) \in A_1 \Leftrightarrow (f(v_1), f(v_2)) \in A_2$$

$$label(v) = label(f(v)), \forall v \in V_1.$$

A *subtree repeat* R in a tree T is a set of node tuples $(u_1, v_1), \ldots, (u_{|R|}, v_{|R|})$, such that $\hat{T}(u_1, v_1) = \ldots = \hat{T}(u_{|R|}, v_{|R|})$. We call $|R|$ the repetition *frequency* of R. If $|R| = 1$ we say that the particular subtree does not repeat. An *overlapping* subtree repeat is a subtree repeat R, where at least one node v is contained in all $|R|$ trees. If no such v exists, we call it a *non-overlapping* subtree repeat. A *total repeat* R is a subtree repeat that contains all nodes in T, that is, $R = \{(u_1, u_1), \ldots, (u_{|R|}, u_{|R|})\}$. See Fig. 1 in this regard.

In the following, we consider the problem of computing all such subtree repeats of an unrooted tree T.

3 Algorithm

The algorithm works in two stages: the forward/non-overlapping stage and the backward/overlapping stage. The forward stage finds all non-overlapping subtree repeats of some tree T. The backward stage uses the identifiers assigned during the forward stage to detect all overlapping subtree repeats, including total repeats.

3.1 The forward/non-overlapping stage

We initially present a brief description of the algorithmic steps. Thereafter, we provide a formal description of each step in Algorithm 1.

In the following, we identify each node in the tree by a unique integer in the range of 1 to $|T|$. Such a unique integer labeling can be obtained, for instance, by a pre- or post-order tree traversal.

The basic idea of the algorithm can be explained by the following steps:

1. Partition nodes by height.
2. Assign a unique identifier to each label in Σ.
3. For each height level starting from 0 (the leaves).
   - i For each node v of the current height level construct a string containing the identifier of the label of v and the identifiers of the subtrees that are attached to v.
   - ii For each such string, sort the identifiers within the string.
   - iii Lexicographically sort the strings (for the current height level).
   - iv Find non-overlapping subtree repeats as identical adjacent strings in the lexicographically sorted sequence of strings.
   - v Assign unique identifiers to each set of repeating subtrees (equivalence class).

We will explain each step by referring to the corresponding lines in Algorithm 1.

Partitioning the nodes according to their height requires time linear with respect to the size of the tree, and is described in line 2 of Algorithm 1. This is done using an array H of queues, where $H[i]$, for all $0 \leq i \leq \lfloor d(T)/2 \rfloor$, contains all nodes of height i. Thereafter, we assign a unique identifier to each label in Σ in lines 3-7. The main loop of the algorithm starts at line 8 and processes the nodes at each height level starting bottom-up from the leaves towards the central points. The main loop consists of four steps. First, a string is constructed for each node v which comprises the identifier for the label at v followed by the identifiers assigned to $u_1, u_2, \ldots, u_{c_v}$. The identifiers $u_1, u_2, \ldots, u_{c_v}$ represent the subtrees $\hat{T}(u_1), \hat{T}(u_2), \ldots, \hat{T}(u_{c_v})$, where $u_1, u_2, \ldots, u_{c_v}$ are the children of v (lines 11-16). Assume that this particular step constructs k strings $s_1, s_2, \ldots, s_k$.

In the next step, we sort the identifiers within each string. To obtain this sorting in linear time, we first need to remap individual identifiers contained as letters in those strings to the range $[1, m]$. Here, m is the number of unique identifiers in the strings constructed for this particular height, and the following

---

Algorithm 1: FORWARD-STAGE

Input : Unrooted tree $T = (V, E)$ labeled from Σ
Output: Sets $\mathcal{R}_{reps}$ of non-overlapping subtree repeats of T

---

1 ▷ Partition tree nodes by height
2 **for** *all* $v \in V$ **do** ENQUEUE($H[h(v)], v$)
3 $cnt \leftarrow 0$
4 ▷ Assign a number from 1 to $|\Sigma|$ to each label
5 **for** *all labels* $\ell \in \Sigma$ **do**
6 $cnt \leftarrow cnt + 1$
7 $L[\ell] \leftarrow cnt$

8 ▷ Compute subtree repeats
9 $reps \leftarrow 0$
10 **for** $i \leftarrow 0$ **to** $\lfloor d(T)/2 \rfloor$ **do**
11 $S \leftarrow \emptyset$
12   ▷ Construct a string of numbers for each node v and its children
13   **foreach** $v \in H[i]$ **do**
14   Let $children(v) = \{ u \mid \{u, v\} \in E \} \setminus \{parent(v)\}$ and $c_v = |children(v)|$
15 $s_v \leftarrow L[label(v)]K[u_1]K[u_2]\ldots K[u_{c_v}]$, if $children(v) = \{u_1, u_2, \ldots, u_{c_v}\}$
16 $S \leftarrow S \cup \{s_v\}$

17   ▷ Remap numbers $[1, |T| + |\Sigma|)$ to $[1, |H[i]| + \sum_{v \in H[i]} c_v]$
18 $R \leftarrow$ REMAP(S)
19   ▷ Bucket sort strings
20   Bucket sort the (unique) numbers of all strings in R.
21   Let R' be the set of individually sorted strings that have been extracted from the respective sorted list from the previous step.
22   Lexicographically sort the strings in R' using radix sort and obtain a sorted list R'' of strings $r_1, r_2, \ldots, r_{|R''|}$.
23   Let each r_i be of the form $k_1^i k_i^2 \ldots k_{|r_i|}^i$ and the corresponding, original unsorted string s_i of the form $L[v_1^i]K[v_2^i]\ldots K[v_{|r_i|}^i]$.
24 $reps \leftarrow reps + 1$
25 $\mathcal{R}_{reps} \leftarrow \{(v_1^1, parent(v_1^1))\}$
26 $K[v_1^1] \leftarrow reps + cnt$
27   **for** $j \leftarrow 2$ **to** k **do**
28   **if** $r_j = r_{j-1}$ **then**
29 $\mathcal{R}_{reps} \leftarrow \mathcal{R}_{reps} \cup \{(v_1^j, parent(v_1^j))\}$
30   **else**
31 $reps \leftarrow reps + 1$
32 $\mathcal{R}_{reps} \leftarrow \{(v_1^j, parent(v_1^j))\}$
33 $K[v_1^j] \leftarrow reps + cnt$

---

property holds: $m \leq \sum_{i=1}^{k} |s_i|$. We then apply a bucket sort to these remapped identifiers and reconstruct the ordered strings $r_1, r_2, \ldots, r_k$ (lines 17-20).

  The next step for the current height level is to find the subtree repeats as identical strings. To achieve this, we lexicographically sort the ordered strings

$r_1, r_2, \ldots, r_k$ (line 22), and check neighboring strings for equivalence (lines 23-33). For each equivalence class $\mathcal{R}_i$ we choose a new, unique identifier, that is assigned to the root nodes of all the subtrees in that class (lines 26 and 33). Finally, each set $\mathcal{R}_i$ contains exactly the tuples of those nodes that are the roots of a particular non-overlapping subtree repeat of T and their respective parents.

Remapping from $\mathcal{D}_1 = [1, |T| + |\Sigma|)$ to $\mathcal{D}_2 = [1, |H[i]| + \sum_{v \in H[i]} c_v]$ can be done using an array A of size $|T| + |\Sigma|$, a counter m, and a queue Q. We read the numbers of the strings one by one. If a number x from domain $\mathcal{D}_1$ is read for the first time, we increase the counter m by one, set $A[x] := m$, and place m in Q. Subsequently, we replace x by m in the string. In case a number x has already been read, that is, $A[x] \neq 0$, we replace x by $A[x]$ in the string. When the remapping step is completed, only the altered positions in array A will be cleaned up, by traversing the elements of Q.

Theorem 1 (Correctness). *Given an unrooted tree T, Algorithm 1 correctly computes all non-overlapping subtree repeats.*

Proof. First note that if any two subtrees $\hat{T}_1$ and $\hat{T}_2$ are repeats of each other, they must, by definition, be of the same height. So the algorithm is correct in only comparing trees of the same height. Additionally, non-overlapping subtrees repeats of a tree T can only be of height $\lfloor d(T)/2 \rfloor$ or less, where $d(T)$ is the diameter of T. Therefore, the algorithm is correct in stopping after processing all $\lfloor d(T)/2 \rfloor + 1$ height classes, in order to extract all the non-overlapping subtree repeats. Since the algorithm only extracts non-overlapping repeats, we define repeats to mean non-overlapping repeats for the rest of this proof. In addition, for simplicity, we consider the rooted version of T for the rest of this proof.

We show that the algorithm correctly computes all repeats for a tree of any height by induction. For the base case we consider an arbitrary tree of height 1 (trees with height 0 are trivial). Any tree of height 1 only has the root node and any number of leafs attached to it. At the root we can never find a subtree repeat, so we only need to consider the next lower (height) level, that is, the leaf nodes. Any two leafs with identical labels will, by construction of the algorithm, be assigned the same identifiers and thus be correctly recognized as repeats of each other.

Now, assume that all (sub)trees of height $m-1$ have correctly been assigned with identifiers by the algorithm *and* that they are identical for two (sub)trees *iff* they are unordered repeats of each other.

Consider an arbitrary tree of height $m+1$. The number of repeats for the tree spanned from the root (node r) is always one (the whole tree). Now consider the subtrees of height m. The root of any subtree of height m must be a child of r. For any child of r that induces a tree of height smaller than m, all repeats have already been correctly calculated according to our assumption.

Two (sub)trees are repeats of each other *iff* the two roots have the same label and there is a one-to-one mapping from subtrees induced by children of the root of one tree to topologically equivalent subtrees induced by children of the root of the second tree. By the induction hypothesis, all such topologically equivalent

subtrees of height $m - 1$ or smaller have already been assigned identifiers that are unique for each equivalence class. Thus, deciding whether two subtrees are repeats of each other can be done by comparing the root labels and the corresponding identifiers of their children, which is exactly the process described in the algorithm. The approach used in the algorithm correctly identifies identically labeled strings since the order of identifiers has been sorted for a given height class. Thus the algorithm finds all repeats of height m (and $m + 1$ at the root). □

Theorem 2 (Complexity). *Algorithm 1 runs in time and space $\mathcal{O}(|T|)$.*

Proof. We prove the linearity of the algorithm by analyzing each of the steps in the outline of the algorithm. Steps 1 and 2 are trivial and can be computed in $|T|$ and $|\Sigma|$ steps, respectively. Notice that $|\Sigma| \leq |T|$.

The main for loop visits each node of T once. For each node v a string s_v is constructed which contains the identifier of the label of v and the identifiers assigned to the child nodes of v. Thus, each node is visited at most twice: once as parent and once as child. This leads to $2n - 1$ node traversals, where n is the number of nodes of T, since the root node is the only node that is visited exactly once. The constructed strings for a height level i are composed of the nodes in $H[i]$ and their respective children. In total we have $c(i) := \sum_{v \in H[i]} c_v$ child nodes at a height level i, where c_v is the number of children of node v. Therefore, the total size of all constructed strings for a particular height level i is $|H[i]| + c(i)$. Step 3ii runs in linear time with respect to the number of nodes at each height level i *and* their children. This is because the remapping is computed in linear time with respect to $|H[i]| + c(i)$. By the remapping, we ensure that the identifiers in each string are within the range of 1 to $|H[i]| + c(i)$. Using bucket sort we can then sort the remapped identifiers in time $|H[i]| + c(i)$ for each height level i. Consequently, the identifiers in each string can be sorted in time $|H[i]| + c(i)$ by traversing the sorted list of identifiers and positioning the respective identifier in the corresponding string on a first-read-first-place basis. This requires additional space $|H[i]| + c(i)$ to keep track which remapped identifier corresponds to which strings.

After remapping and sorting the strings, finding identical strings as repeats requires a lexicographical sorting of the strings. Strings that are identical form classes of repeats. Lexicographical sorting (using radix sort) requires time $\mathcal{O}(|H[i]| + c(i))$ and at most space for storing $|T| + |\Sigma|$ elements since the identifiers are in the range of 1 to $|T| + |\Sigma|$. This memory space needs to be allocated only once. Moreover, the elements that have been used are cleared/cleaned-up at each step via a queue as explained for the remapping function.

By summing over all height levels we obtain $\sum_{i=0}^{\lfloor d(T)/2 \rfloor} (|H[i]| + c(i)) = 2n - 1$. Thus the total time over all height levels for each step described in the loop is $\mathcal{O}(|T|)$. The overall time and space complexity of the algorithm is thus $\mathcal{O}(|T|)$. □

We conclude this section with an example demonstrating Algorithm 1. Consider the tree T from Fig. 2. The superscript indices denote the number associated
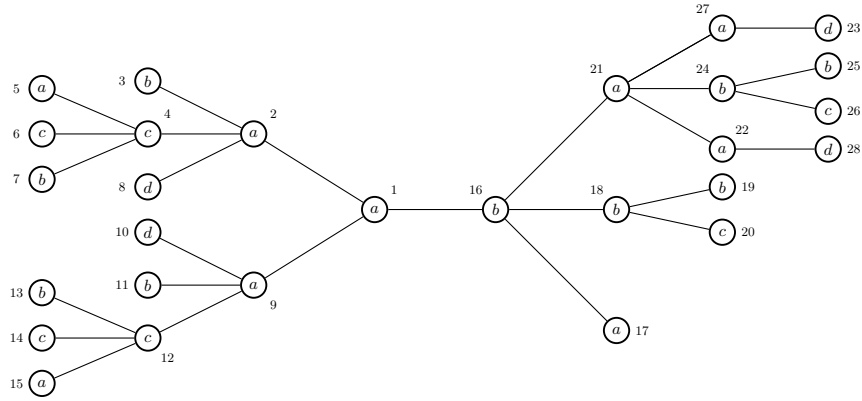
Fig. 2. Graphical representation of tree T. The superscript indices denote the unique identifier assigned to each node by traversing T

with each node, which, in this particular example, correspond to a pre-order traversal of $\hat{T}(T)$ by designating node 1 as the root. Lines 1-2 partition the nodes of T in $\lfloor d(T)/2 \rfloor + 1$ sets according to their height. The sets $H[0] = \{3, 5, 6, 7, 8, 10, 11, 13, 14, 15, 17, 19, 20, 23, 25, 26, 28\}$, $H[1] = \{4, 12, 18, 22, 24, 27\}$, $H[2] = \{2, 9, 21\}$ and $H[3] = \{1, 16\}$ are created. Lines 5-7 create a mapping between labels and numbers. $L[a] = 1$, $L[b] = 2$, $L[c] = 3$, and $L[d] = 4$. Table 1 shows the state of lists S, R, R', R'' during the computation of the main loop of Algorithm 1 for each height level, where S is the list of string identifiers, R is the list of remapped identifiers, R' is the list of individually sorted remapped identifiers, and R'' is the list R' lexicographically sorted. Fig. 3 depicts tree T with the respective identifiers for each node as assigned by Algorithm 1.

3.2 The backward/overlapping stage

Definition 1 (Sibling repeat). *Given an unrooted tree T, two equal subtrees of $\hat{T}(T)$ whose roots have the same parent are called a* sibling repeat.

Definition 2 (Child repeat). *Given an unrooted tree T, two subtrees of $\hat{T}(T)$ whose root's have the same identifiers and whose root's respective parents are roots of trees in the same sibling or child repeat, are called a* child repeat.

Note that with these definitions we get that two trees with roots u and v respectively, are child or sibling repeats of each other *iff* the unique path between nodes u and v is symmetrical with respect to the node labels of the nodes traversed on the path. Also note, that child repeats and sibling repeats can occur in the same repeat class; it is merely a property shared between two (or more) trees.

The two following lemmas illustrate why it is necessary and sufficient to know the identifiers from the forward stage to compute all overlapping subtree repeats.

Height	Step	Process	Repeats
0	Strings: S	$2, 1, 3, 2, 4, 4, 2, 2, 3, 1, 1, 2, 3, 4, 2, 3, 4$	$\mathcal{R}_1 = \{3, 7, 11, 13, 19, 25\}$
	Remapping: R	$1, 2, 3, 1, 4, 4, 1, 1, 3, 2, 2, 1, 3, 4, 1, 3, 4$	$\mathcal{R}_2 = \{5, 15, 17\}$
	Sorting: R'	$1, 2, 3, 1, 4, 4, 1, 1, 3, 2, 2, 1, 3, 4, 1, 3, 4$	$\mathcal{R}_3 = \{6, 14, 20, 26\}$
	Repeats: R''	$\underbrace{1, 1, 1, 1, 1, 1}_{5}, \underbrace{2, 2, 2}_{6}, \underbrace{3, 3, 3, 3}_{7}, \underbrace{4, 4, 4, 4}_{8}$	$\mathcal{R}_4 = \{8, 10, 23, 28\}$
1	Strings: S	3 6 7 5, 3 5 7 6, 2 5 7, 1 8, 2 5 7, 1 8	
	Remapping: R	1 2 3 4, 1 4 3 2, 5 4 3, 6 7, 5 4 3, 6 7	$\mathcal{R}_7 = \{22, 27\}$
	Sorting: R'	1 2 3 4, 1 2 3 4, 3 4 5, 6 7, 3 4 5, 6 7	$\mathcal{R}_5 = \{4, 12\}$
	Repeats: R''	$\underbrace{1\ 2\ 3\ 4, 1\ 2\ 3\ 4}_{9}, \underbrace{3\ 4\ 5, 3\ 4\ 5}_{10}, \underbrace{6\ 7, 6\ 7}_{11}$	$\mathcal{R}_6 = \{18, 24\}$
2	Strings: S	1 5 9 8, 1 8 5 9, 1 11 10 11	
	Remapping: R	1 2 3 4, 1 4 2 3, 1 5 6 5	$\mathcal{R}_8 = \{2, 9\}$
	Sorting: R'	1 2 3 4, 1 2 3 4, 1 5 5 6	$\mathcal{R}_9 = \{21\}$
	Repeats: R''	$\underbrace{1\ 2\ 3\ 4, 1\ 2\ 3\ 4}_{12}, \underbrace{1\ 5\ 5\ 6}_{13}$	
3	Strings: S	2 6 10 13, 1 12 12	
	Remapping: R	1 2 3 4, 5 6 6	$\mathcal{R}_{10} = \{16\}$
	Sorting: R'	1 2 3 4, 5 6 6	$\mathcal{R}_{11} = \{1\}$
	Repeats: R''	$\underbrace{1\ 2\ 3\ 4}_{14}, \underbrace{5\ 6\ 6}_{15}$	

Table 1. State of lists S, R, R', R'' for each height level and resulting sets $\mathcal{R}_{reps}$ of non-overlapping subtree repeats
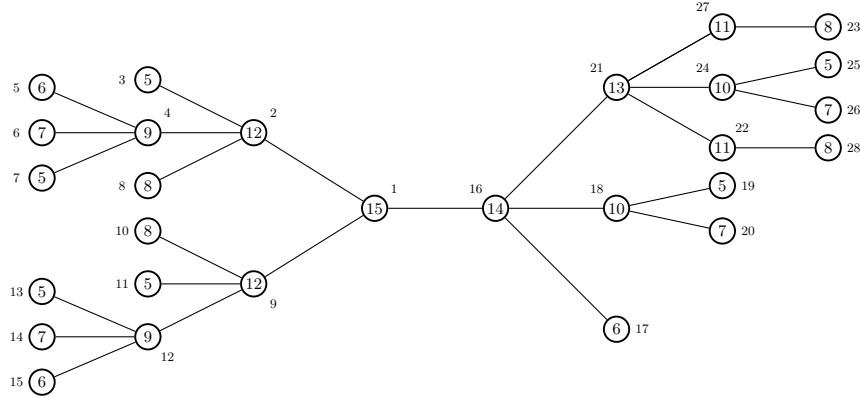

Fig. 3. Graphical representation of tree T with the associated identifier for each node as assigned by Algorithm 1

Lemma 1 (Sufficient conditions). *Let r be the parent of u and v, where u and v are roots of a sibling repeat. Then the trees $\hat{T}(u, u)$ and $\hat{T}(v, v)$ are elements of the same total repeat. The trees $\hat{T}(r, u)$ and $\hat{T}(r, v)$ are elements of the same overlapping subtree repeat.*

Let u and v be roots of a child repeat. Further let r_u and r_v be the parents of u and v, respectively. Then the trees $\hat{T}(u, u)$ and $\hat{T}(v, v)$ are elements of the same total repeat, and the trees $\hat{T}(r_u, u)$ and $\hat{T}(r_v, v)$ are elements of the same overlapping subtree repeat.

Proof. Trivial, by inspection; see Fig. 2. □

In Fig. 2, the trees $\hat{T}(2, 1)$ and $\hat{T}(9, 1)$ form a sibling repeat, thus the trees $\hat{T}(4, 2)$ and $\hat{T}(12, 9)$ form a child repeat. From the sibling repeat, we get that $\hat{T}(2, 2)$ and $\hat{T}(9, 9)$ are elements of a total repeat, while $\hat{T}(1, 2)$ and $\hat{T}(1, 9)$ are within the same overlapping repeat. Analogously, for the child repeat we get the trees $\hat{T}(4, 4)$ and $\hat{T}(12, 12)$ as total repeats and $\{(2, 4), (9, 12)\}$ as an overlapping repeat.

Note that Lemma 1 implies that all nodes of a subtree that is element of an overlapping subtree repeat with repetition frequency $|R|$ are roots of trees in overlapping repeat classes of frequency at least $|R|$.

Lemma 2 (Necessary conditions). *Any two trees that are elements of a total repeat must have been assigned the same identifiers at their respective roots during the forward stage, and be rooted in roots of either sibling or child repeats.*

Any two trees that are elements of an overlapping subtree repeat, but not of a total repeat, must have been assigned the same identifiers at their respective roots during the forward stage, and be rooted in parents of roots of either sibling or child repeats.

Proof. We first look at the case of total repeats. Let $\hat{T}(u, u) = \hat{T}(v, v)$. We now consider the unique path p between u and v. Obviously, for equality among these two trees to hold, the path must be symmetrical, which by recursion implies that u and v are roots of either sibling or child repeats; see Fig. 4.

The case of other overlapping subtree repeats works just the same. Let $\hat{T}(r_u, u) = \hat{T}(r_v, v)$ not be total, but overlapping subtree repeat. These trees are obtained by removing a single edge from the tree: (r_u, u) and (r_v, v), respectively. Let p be the path between u and v. Since the trees are elements of an overlapping subtree repeat, r_u and r_v must lie on this path. Additionally, since r_u and r_v are on the path from u to v, $h(v) = h(u)$, and any tree is acyclic, then r_u and r_v must be closer to the central points than u and v, respectively. Since there is an edge connecting r_u with u and r_v with v this means that r_u and r_v are parents of u and v, respectively. Again, the path p is symmetrical with respect to the node labels of nodes along the path, so u and v are roots of either sibling or child repeats. □

Given these two lemmas, we can compute all overlapping subtree repeats by checking for sibling and child repeats. This can be done by comparing the identifiers assigned to nodes in the forward stage. The actual procedure of computing all overlapping subtree repeats is described in Algorithm 2. Algorithm 2 takes as input an unrooted tree T that has been processed by Algorithm 1; i.e. each node of tree T has already been assigned an identifier according to its non-overlapping repeat class.
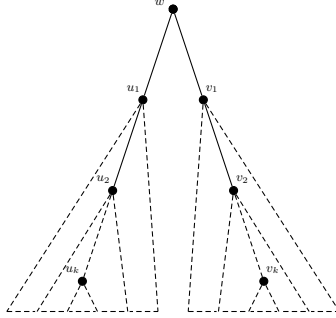
Fig. 4. $T(v_2, v_k) = T(u_2, u_k)$ is an overlapping repeat *iff* $T(u_k, u_2) = T(v_k, v_2)$ is a child repeat, which is true *iff identifier*$(u_k) = identifier(v_k)$, *identifier*$(u_2) = identifier(v_2)$, *identifier*$(u_1) = identifier(v_1)$

First, the algorithm considers the rooted version of T, that is $\hat{T}(T)$. This is done since many operations and definitions rely on $\hat{T}(T)$. Next, we define a queue Q, whose elements are sets of nodes. Initially, Q contains only the set containing the root node of $\hat{T}(T)$ (line 2). Processing Q is done by dequeuing a single set of nodes at a time (lines 5-16). For a given set U of Q, the algorithm creates a set I containing the identifiers of children of all the nodes in U. Then, the algorithm remaps these identifiers to the range of $[1, |I|]$ constructing a new set I' (line 12). Then, we construct a list C of tuples, such that each tuple contains the remapped identifier of a child and the corresponding node. Therefore, we can use bucket sort to sort these tuples by the remapped identifiers in time linear in the cardinality of I.

We are now in a position to apply Lemmas 1 and 2. By Lemma 2, finding sibling and child repeats is done by creating sets of nodes with equivalent identifiers in C (line 18). This can be easily done due to the sorting part of the algorithm. These sets are then enqueued in Q, and, by Lemma 1 and 2, all resulting subtree repeats (overlapping and total) are, thus, created (lines 21-22). Hence we immediately obtain the following result.

Theorem 3 (Correctness). *Given an unrooted tree T with identifiers assigned by Algorithm 1, Algorithm 2 correctly computes all overlapping subtree repeats, including total repeats.*

Algorithm 2 enqueues each node of T once. For each enqueued node, a constant number of operations is performed. Therefore we get the following result.

Theorem 4 (Complexity). *Algorithm 2 runs in time and space $\mathcal{O}(|T|)$.*

Algorithm 2: BACKWARD-STAGE

> **Input** : Unrooted tree $T = (V, E)$ labeled from Σ with identifiers assigned by
>   Algorithm 1
> **Output**: Sets $\mathcal{R}'_{reps}$ of overlapping subtree repeats of T
---
1 $\triangleright$ Initialize queue Q with the root node r of $\hat{T}(T)$
2 ENQUEUE$(Q, \{r\})$
3 $\triangleright$ Compute overlapping subtree repeats
4 **while** QUEUE-NOT-EMPTY(Q) **do**
5 $U \leftarrow$ DEQUEUE(Q)
6 $\triangleright$ Get the identifiers of the children of the nodes in U
7   Let $cod(U)$ be the cumulated out degree of all the nodes in U
8   Let $children(U) = \{u_1, u_2, \ldots, u_{cod(U)}\}$ be the children of the nodes in U
9   Let $ids(children(U)) = \{i_1, i_2, \ldots, i_{cod(U)}\}$ be the identifiers of
       $\{u_1, u_2, \ldots, u_{cod(U)}\}$
10 $I \leftarrow ids(children(U))$
11 $\triangleright$ Remap numbers $[1, |T| + |\Sigma|)$ to $[1, |I|]$
12 $I' \leftarrow$ REMAP(I)
13   Let $I' = \{i'_1, i'_2, \ldots, i'_{cod(U)}\}$ be the remapped identifiers of
       $\{u_1, u_2, \ldots, u_{cod(U)}\}$
14   Let $C = <i'_1, u_1>, <i'_2, u_2>, \ldots, <i'_{cod(U)}, u_{cod(U)}) >$ be a list of tuples
15 $\triangleright$ Bucket sort the remapped identifiers
16   Bucket sort the list C by $i'_1, i'_2, \ldots, i'_{cod(U)}$.
17 $\triangleright$ Extract the equivalence classes
18   **foreach** $E = \{v_1, v_2, \ldots, v_k\}$ *of nodes with equivalent identifiers in C* **do**
19   ENQUEUE(Q, E)
20   **for** $i \leftarrow 1$ **to** k **do**
21 $\mathcal{R}'_{reps} \leftarrow \mathcal{R}'_{reps} \cup \{(parent(v_i), v_i)\}$
22 $\mathcal{R}'_{reps+1} \leftarrow \mathcal{R}'_{reps+1} \cup \{(v_i, v_i)\}$
23 $reps \leftarrow reps + 2$
---

4   Final remarks

We presented a simple and time-optimal algorithm for computing all full subtree repeats in unrooted unordered labeled trees; and showed that the running time of our method is linear with respect to the size of the input tree.

The presented algorithm can easily be modified to operate on trees that do not satisfy some or any of the aforementioned assumptions on the tree structure.

– *Rooted trees*: In a rooted tree $\hat{T}$, only non-overlapping repeats can occur. Therefore it is sufficient to apply Algorithm 1 with the following modifications: first, we define $\hat{T}(\hat{T}) := \hat{T}$; second, the main for loop must iterate over the height of $\hat{T}$, instead of depending on its diameter.
– *Ordered trees*: If for a node the order of its adjacent nodes is relevant, i.e. the tree is ordered, the bucket sort procedures in Algorithms 1 and 2 must

be omitted. Additionally, sibling repeats must not be merged in line 19 of Algorithm 2 but rather be enqueued separately.
– *Unlabeled trees*: Trivially, an unlabeled tree can be seen as a labeled tree with a single uniform symbol assigned to all nodes.

Algorithm 1 can also be used to compute subtree repeats over a *forest* of rooted unordered trees. The method is the same as for the case of a single tree. The method reports all subtree repeats by clustering the identifiers of equal subtrees from all trees in the forest into an equivalence class. The correctness of this approach can be trivially obtained by connecting the roots of all trees in the forest with a virtual root node, and applying the algorithm to this single tree.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison Wesley, 2 edn. (2006)
2. Barstow, D.R., Shrobe, H.E., Sandewall, E.: Interactive Programming Environments. McGraw-Hill, Inc. (1984)
3. Chor, B., Tuller, T.: Finding a maximum likelihood tree is hard. Journal of ACM 53(5), 722–744 (2006)
4. Christou, M., Crochemore, M., Flouri, T., Iliopoulos, C.S., Janoušek, J., Melichar, B., Pissis, S.P.: Computing all subtree repeats in ordered trees. Information Processing Letters 112(24), 958–962 (2012)
5. Christou, M., Crochemore, M., Flouri, T., Iliopoulos, C.S., Janoušek, J., Melichar, B., Pissis, S.P.: Computing all subtree repeats in ordered ranked trees. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) String Processing and Information Retrieval. Lecture Notes in Computer Science, vol. 7024, pp. 338–343. Springer (2011)
6. Felsenstein, J.: Evolutionary trees from DNA sequences: a maximum likelihood approach. Journal of molecular evolution 17(6), 368–376 (1981)
7. Felsenstein, J.: Inferring phylogenies. Sinauer Associates (2003)
8. Ferdinand, C., Seidl, H., Wilhelm, R.: Tree automata for code selection. Acta Inf. 31, 741–760 (1994)
9. Guindon, S., Dufayard, J.F., Lefort, V., Anisimova, M., Hordijk, W., Gascuel, O.: New Algorithms and Methods to Estimate Maximum-Likelihood Phylogenies: Assessing the Performance of PhyML 3.0. Systematic Biology 59(3), 307–321 (2010)
10. Harary, F.: Graph Theory. Addison Wesley Publishing Company (1994)
11. Hoffmann, C.M., O'Donnell, M.J.: Programming with equations. ACM Trans. Program. Lang. Syst. 4, 83–112 (1982)
12. Hudak, P.: Conception, evolution, and application of functional programming languages. ACM Computing Surveys 21, 359–411 (1989)
13. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebra. In: Leech, J. (ed.) Computational problems in abstract algebra, pp. 263–297. Pergamon Press (1970)
14. Mauri, G., Pavesi, G.: Algorithms for pattern matching and discovery in RNA secondary structure. Theoretical Computer Science 335(1), 29–51 (2005)
15. Stamatakis, A.: RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. Bioinformatics 22(21), 2688–2690 (2006)
16. Yang, Z.: Computational Molecular Evolution. Oxford University Press, Oxford (2006)