Near-Optimal Algorithm for Constructing Greedy Consensus Tree

Hongxun Wu
Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China
wuhx18@mails.tsinghua.edu.cn

Abstract

In biology, phylogenetic trees are important tools for describing evolutionary relations, but various data sources may result in conflicting phylogenetic trees. To summarize these conflicting phylogenetic trees, consensus tree methods take k conflicting phylogenetic trees (each with n leaves) as input and output a single phylogenetic tree as consensus.

Among the consensus tree methods, a widely used method is the greedy consensus tree. The previous fastest algorithms for constructing a greedy consensus tree have time complexity $\tilde{O}(kn^{1.5})$ [Gawrychowski, Landau, Sung, Weimann 2018] and $\tilde{O}(k^2n)$ [Sung 2019] respectively. In this paper, we improve the running time to $\tilde{O}(kn)$. Since k input trees have $\Theta(kn)$ nodes in total, our algorithm is optimal up to polylogarithmic factors.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases phylogenetic trees, greedy consensus trees, splay tree

Digital Object Identifier 10.4230/LIPIcs.ICALP.2020.105

Category Track A: Algorithms, Complexity and Games

Acknowledgements We want to thank anonymous reviewers for many helpful comments.

1 Introduction

The problem of constructing consensus trees arises from bioinformatics. In biology, phylogenetic trees describe the biological evolutionary relations between species. But the phylogenetic trees from different biological data may conflict with each other. As mentioned in [22], even from the same data set, when certain resampling techniques are used, we could still get many different phylogenetic trees. This problem has long been studied, to name a few [1, 14, 28, 17, 13, 12, 18, 5].

Motivated by this, consensus tree methods were proposed [1] to summarize these phylogenetic trees into a single phylogenetic tree, which is viewed as the consensus of these phylogenetic trees and is called the consensus tree. Since then, many different consensus tree methods were proposed. As mentioned in [22], the majority rule consensus tree [28], the loose consensus tree [8], and the greedy consensus tree [9] are the most frequently used consensus trees.

As discussed in [11], while increasing the number of phylogenetic trees in the input, the greedy consensus tree converges faster than majority rule consensus tree and R^* consensus tree. Although the greedy consensus tree is not a consistent estimator, the region of parameter space in which greedy consensus tree fails is relatively small, hence greedy consensus tree offers more robustness [11]. The greedy consensus tree method is implemented in many software packages, such as PHYLIP [15], PAUP* [40], MrBayes [32], RAxML [37], and also widely used in numerous works in biology [6, 7, 11, 24, 26, 27, 30, 31, 33, 34, 36, 38].

For most consensus tree methods, optimal or near-optimal algorithms for construction have been found. One exception is the greedy consensus tree (See Table 1). For greedy consensus tree construction, the naïve algorithm takes $\tilde{O}(kn^3)$ time [9]. Then it was improved to $O(kn^2)$ time by [22]. Recently there are $\tilde{O}(kn^{1.5})$ [16] and $\tilde{O}(k^2n)$ [39] algorithms proposed for it.

Table 1 Running time of construction algorithms for different consensus tree methods.

Consensus tree method	Running time	Reference
Adam's consensus tree	$O(kn \log n)$	[19]
Strict consensus tree	$O(kn)$	[10]
Loose consensus tree	$O(kn)$	[22]
Frequency difference consensus tree	$O(kn \log^2 n)$	[16]
Majority-rule consensus tree	$O(kn \log k)$, Randomized $O(kn)$	[4, 22]
Majority-rule (+) consensus tree	$O(kn)$	[21]
Local consensus tree	$O(kn^3)$	[25, 20]
R^* consensus tree	$O(n^2 \log^{k+2} n)$	[23]
Greedy consensus tree	$O(kn^{1.5})$, $O(k^2 n)$	[16, 39]

In this paper, we present a near-optimal $\tilde{O}(kn)$ time algorithm for greedy consensus tree construction.

▶ **Theorem 1.** *Greedy consensus tree of k phylogenetic trees for n species can be constructed in $\tilde{O}(kn)$ time.*

High Level Idea

The previous $\tilde{O}(kn^{1.5})$ algorithm [16] builds the consensus tree by dynamically adding nodes to it. To find out the position to add a new node, they need to support least common ancestor query on the consensus tree. Since the consensus tree is dynamically changing, answering such queries is time-consuming and becomes the bottleneck of this previous algorithm. Motivated by it, we came up with an alternative approach that only requires least common ancestor query on the phylogenetic trees in the input. Since these are static trees, such queries can be efficiently answered. This leads to our improved algorithm.

2 Preliminaries

2.1 Phylogenetic Tree

Phylogenetic trees represent the evolution of species. Different leaves of a phylogenetic tree represent different species. From biological data, we can infer that some of these species share common ancestors. These common ancestors are represented by the nonleaf nodes, also called inner nodes. Each inner node represents the common ancestor of all leaves in its subtree.

Formally, a tree with n leaves is leaf-labeled if and only if its n leaves have distinct labels from 1 to n. A phylogenetic tree T is a rooted, unordered, leaf-labeled tree in which every inner node has at least two children. If there is a directed path from node u to node v, u is a descendant of v, and v is an ancestor of u. Thus node u is a descendant of itself. If $u \neq v$ and v is a descendant of u, v is a proper descendant of u, and u is a proper ancestor of v.

The subtree tree of an inner node u is the subtree rooted at u and formed by all its descendants. In the rest of the paper, whenever we say a subtree, we always refer to the subtree of an inner node.

Cluster and Signature

For inner node v in phylogenetic tree T, we define $L(v)$ to be the set of species within its subtree. Namely, $L(v) = \{x \in [n] : \text{There is a leaf labeled } x \text{ in the subtree of } v\}$. The set $L(v)$ is called a cluster.

Based on $L(v)$, we define the signature of phylogenetic tree T to be the set of all clusters in it. Namely, $\text{sign}(T) = \{L(v) : v \text{ is an inner node in } T\}$.

▶ **Observation 2.** *The signature of a phylogenetic tree completely captures its structure. Namely, given sign(T), the phylogenetic tree T is uniquely determined.*

More specifically, suppose the cluster $c_1 \in sign(T)$ corresponds to node u on T. Namely, $c_1 = L(u)$. Let $par(u)$ denote the parent of u on T. $L(par(u))$ is the smallest cluster $c_2 \in sign(T)$ such that $c_1 \subset c_2$. This determines the parent of each node.

Consistency

But not every set S of clusters can be the signature of a phylogenetic tree. S is a valid signature if and only if it is a laminar family. In this case, we say S is consistent. Namely, S is consistent if and only if for every pair of distinct clusters $c_1, c_2 \in S$, one of the following holds:

- $c_1 \cap c_2 = \emptyset$
- $c_1 \subset c_2$
- $c_1 \supset c_2$

We say a cluster c is consistent with S if and only if $S \cup \{c\}$ is consistent.

2.2 Greedy Consensus Tree

Recall that the greedy consensus tree method takes k phylogenetic trees $T_1, T_2, \cdots, T_k$ as inputs. These k trees are over the same set of n species. Its objective is to output a single phylogenetic tree to be the consensus tree.

Definition

As its name suggests, a greedy consensus tree is defined as the output of a simple greedy algorithm. Let $\mathcal{F}$ be the set of all clusters that have appeared in $T_1, T_2, \cdots, T_k$. The frequency $f(c)$ of a cluster c is the number of times c appears in $\mathcal{F}$.

Let S denote the signature of the current consensus tree. Initially, the consensus tree only contains a root and n leaves. Thus $S = \{\{1, 2, \ldots, n\}, \{1\}, \{2\}, \ldots, \{n\}\}$. Each time we pick the cluster with the highest frequency in $\mathcal{F}$ and add it to S if they are consistent. From the tree point of view, we are adding a new inner node to the consensus tree. For the details, see Algorithm 1.

Algorithm 1 Greedy consensus tree.

1: Initially signature $S \leftarrow \{\{1, 2, \ldots, n\}, \{1\}, \{2\}, \ldots, \{n\}\}$
2: $\mathcal{F} \leftarrow \text{sign}(T_1) \cup \text{sign}(T_2) \cup \cdots \cup \text{sign}(T_k)$
3: For all clusters $c \in \mathcal{F}$, count its frequency $f(c) \leftarrow |\{i | c \in \text{sign}(T_i)\}|$
4: **while** $\mathcal{F} \neq \emptyset$ **do**
5: Pick $c_0 \leftarrow \arg\max_{c \in \mathcal{F}} f(c)$ with the highest frequency (ties are broken arbitrarily)
6: **if** c_0 is consistent with S **then** $S \leftarrow S \cup \{c_0\}$
7: $\mathcal{F} \leftarrow \mathcal{F} \backslash \{c_0\}$
8: S is the signature of a greedy consensus tree

Since at Line 5, Algorithm 1, ties are broken arbitrarily, the output of the algorithm may not be unique. There may be more than one greedy consensus tree for a fixed input.

Running Time

Algorithm 1 is a naïve algorithm for constructing greedy consensus trees. Here we discuss this algorithm and its complexity in more detail. It essentially contains two phases:

1. Count the frequency $f(c)$ of clusters and sort them. (Line $1 \sim 3$ of Algorithm 1)
2. Repeatedly run the greedy procedure. (Line $4 \sim 7$ of Algorithm 1)

The first phase is easy to be handled in $\tilde{O}(kn)$ time. Suppose $|\mathcal{F}| = m$. Namely, there are m distinct clusters. For each of them, we assign a unique number in $[m]$ to it as its identifier. For each $u \in T_i$, let $id(u) \in [m]$ be the identifier of $L(u)$. Then, for $u \in T_i$ and $v \in T_j$, $id(u) = id(v)$ if and only if $L(u) = L(v)$.

In [16], they showed that these identifiers can be computed in $O(kn \log^2 n)$ time. To be self-contained, we state it in Lemma 4 and present a short proof here. We first need a data structure for dynamic set equality.

▶ **Lemma 3** (Lemma 1, [16]). *There is a deterministic dynamic set equality structure that supports:*

- *CREATE(s): Create a new empty set s.*
- *ADD(s, x): Add an element x to set s. (create a new set $s \cup \{x\}$ without destroying s)*
- *ID(s): Return the identifier of set s which is a positive integer smaller than the number of sets we have created. Two sets have the same identifier if and only if they are equal.*

Let n denote the size of all sets created. Each operation takes $\tilde{O}(1)$ time.

Proof. We apply the dynamic string equality structure in [29]. It supports the following operations in $\tilde{O}(1)$ time: (1) create a new string with a single character, (2) test if two strings are equal, (3) split a string into two new strings without destroying it, (4) concatenate two strings to form a new string without destroying them, (5) given i, return the i-th character in a string. Modifying a character of a string can be reduced to $O(1)$ split, create, and concatenate operations. Comparing the lexicographical order of two strings can be reduced to $O(\log n)$ split and equality testing operations using binary search. Thus they all take $\tilde{O}(1)$ time.

We encode each set s into a binary string. The i-th bit of the string is 1 if and only if $i \in s$. For empty set, we create a new string with n zeros in beginning. To add an element x to s, we only need to modify the x-th bit in the string which take $\tilde{O}(1)$ time.

To maintain the identifier of each set s, we maintain a global balanced binary search tree of the strings of all sets in their lexicographical order. When a new set is created, we add its string to this balanced binary search tree, and attach a new identifier to it. Since comparing lexicographical order takes $\tilde{O}(1)$ time, maintaining this binary search tree and finding the identifier of a set also takes $\tilde{O}(1)$ time. ◀

▶ **Lemma 4** (Theorem 3, [16]). *The identifier $id(u)$ can be found for every node u of phylogenetic trees $T_1, T_2, \cdots, T_k$ in $\tilde{O}(kn)$ time.*

Proof. We process each tree T_i bottom-up. For each inner node u, we obtain $L(u)$ and its identifier in the dynamic set equality structure by the following procedure:

1. Let v be the children of u with the largest subtree. If v is a leaf, we let s be the singleton $\{\text{label}(v)\}$. Otherwise, let $s = L(v)$ in dynamic set equality structure.
2. We traverse the subtrees of all other children of u and add the leaves we visited into s. Now s equals $L(u)$
3. $id(u) \leftarrow \text{ID}(s)$.

Since each time a node is visited at Step 2, the subtree it belongs to is doubled. Each node can be visited at most $O(\log n)$ times throughout the procedure. Thus this procedure takes $\tilde{O}(kn)$ time in total. ◀

After we get the identifiers $id(u)$, counting the frequency of identifiers and sorting within $\tilde{O}(kn)$ time are straight-forward.

The bottleneck of the naïve algorithm is the second phase. To test whether the signature S of the consensus tree is consistent with c_0, the naïve algorithm enumerates all clusters in S. There can be at most $O(n)$ clusters in S since each of them corresponds to a node in the consensus tree.

For every cluster in S testing whether it is consistent with c_0 takes $O(n)$ time. There are $O(n)$ clusters in S. Thus for each c_0, it takes at most $O(n^2)$ time. There are $O(kn)$ clusters in $\mathcal{F}$ (since each of them corresponds to at least one node in $T_1, T_2, \ldots, T_k$).

So in total, the naïve algorithm takes $O(kn^3)$ time. We will present our algorithm for the second phase in Section 3 and show how to make it efficient in Section 4.

2.3 Data Structures

Our algorithm relies on some classical data structure techniques introduced here.

DFS Sequence

Let T be a rooted tree. The Euler tour of T starts from its root, passing by each edge exactly twice (from opposite directions), and return to the root. We define $E(T)$ to be the Euler tour of T in which we only keep the first occurrence of each node. Or equivalently, $E(T)$ is the sequence produced by the following depth-first search: Initially let the sequence be empty. When we perform a depth-first search on T, each time we visit a node for the first time, we add it to the end of our sequence. In the end, this sequence equals $E(T)$.

▶ **Observation 5.** *Suppose v is a node on T. All nodes within the subtree of v form a continuous interval in $E(T)$.*

For the subtree of v on T, we denote its corresponding interval by $[l_T(v), r_T(v)]$. Here $l_T(v)$ is the position of v in $E(T)$, and $r_T(v)$ is the position of the last node that belongs to the subtree of v in $E(T)$.

Top Tree

A dynamic forest is a set of trees over disjoint sets of nodes that supports dynamic edge connection and deletion. Top tree [3] is a useful data structure for maintaining information in a dynamic forest. k-th ancestor of node u is the ancestor which is higher than u by k edges. When $k = 1$, it is the parent of u.

▶ **Lemma 6** ([3]). *Top tree supports the following operations in $\tilde{O}(1)$ time:*

1. *connect(u, v) : Add an edge connecting nodes u and v that belong to different trees.*
2. *delete(u, v) : Delete the edge connecting nodes u and v.*
3. *lca(u, v) : Return the least common ancestor of u and v.*
4. *ancestor(u, k) : Return the k-th ancestor of u.*

Splay tree

Splay tree [35] is a classical binary search tree maintaining a dynamic sequence. Each element in the sequence has two attributes, its key and value. The elements in a dynamic sequence are arranged in the increasing order of their keys. While keys specify the order of the elements, values are the information related to our queries.

▶ **Lemma 7.** *Let K be an ordered set, and let $G = (S, +)$ be a semigroup. Splay tree maintains n nodes each with its key and value, supporting the following operations:*

1. *$Insert(k, v)$: Insert a new node with key $k \in K$ and value $v \in S$.*
2. *$Delete(k)$: Delete the elements with key $k \in K$.*
3. *$Split(k)$: Return two splay trees T_1, T_2. T_1 contains all nodes whose keys are smaller than k, and T_2 contains all other nodes. The original splay tree is destroyed after the operation.*
4. *$Size()$: Return the number of nodes in the splay tree.*
5. *$Merge(T_1, T_2)$: Merge two splay trees T_1, T_2 where all nodes in T_1 have smaller keys than those in T_2.*
6. *$Sum(k_1, k_2)$: Suppose the values of nodes with keys $k \in [k_1, k_2]$ are $v_1, v_2, \ldots, v_t$ in the order of increasing keys. Then it returns $v_1 + v_2 + \cdots + v_t$.*

Suppose the $+$ operation of semigroup G takes $\tilde{O}(1)$ time. Then each operation here takes $\tilde{O}(1)$ time.

Specifically, it has the following two applications:

- *Let S be the set of integers, and let $+$ be addition. We can answer the summation of a continuous subsequence in $\tilde{O}(1)$ time.*
- *Let S be the set of nodes in a static tree, and let $a + b$ be the least common ancestor of a and b. We can answer the least common ancestor of a continuous subsequence in $\tilde{O}(1)$ time.*

Proof. The original paper [35] for splay tree showed how to handle insert, delete, split, and merge operations when nodes only have keys but no values.

Here at each node, in addition to its key, we also maintain its value and the summation of all values in its subtree. (Here summation refers to the operation of the semigroup) This summation can be calculated from the summation of its children by a $+$ operation. Whenever the children of a node in splay tree changes, we update its summation. Since for each operation, we only change the children of $O(\log n)$ nodes. Update the summation for them takes $O(\log n)$ many $+$ operations only.

To evaluate $Sum(k_1, k_2)$, we first split the splay tree into three trees T_1, T_2, T_3 using two splits. T_1 contains all nodes with keys smaller than k_1. T_2 contains all elements with keys in $[k_1, k_2]$. T_3 contains all other elements. Then we return the summation maintained at the root of T_2, and merge them back.

For $Size()$ query, we can let the value of each node be one and reduce it to the value summation query.

For the applications, since LCA operation is associative, $(S, +)$ is a semigroup. The LCA of two nodes can be answered in $\tilde{O}(1)$ time [2]. Thus each operation takes only $\tilde{O}(1)$ time, and we can answer the least common ancestor of a continuous subsequence by $Sum(k_1, k_2)$.

Similarly, we can answer the summation of a continuous subsequence in $\tilde{O}(1)$ time by $Sum(k_1, k_2)$. ◀

3 Algorithm

In the naïve algorithm, checking whether a cluster c_0 is consistent with signature S takes $O(n^2)$ time. To speed it up, the first step is to find a characterization of consistency that utilizes the tree structure. Here we start with the characterization from the previous $\tilde{O}(kn^{1.5})$ algorithm [16]. Then we develop it into an improved algorithm.

From this section, we will be using the following notations. $\mathcal{T}$ denotes the phylogenetic tree corresponding to signature S, namely our consensus tree. $\mathrm{LCA}_{\mathcal{T}}(c_0)$ is the least common ancestor of all species in cluster c_0 on consensus tree $\mathcal{T}$, while $LCA_i(c_0)$ is that on the input phylogenetic tree T_i. $subtree(v)$ denotes the set of all nodes (both inner nodes and leaves) within the subtree of v.

3.1 Characterization of consistency

In this subsection, the proof of Lemma 8 and 9 are already known from [16], but for clarity, we formally state and prove them here.

To utilize the tree structure, we will focus on consensus tree $\mathcal{T}$ instead of its signature S. The cluster c_0 is consistent with S if and only if for all nodes $u \in \mathcal{T}$, $L(u)$ is consistent with c_0. First, we begin with a lemma that says only those nodes within the subtree of $LCA_{\mathcal{T}}(c_0)$ matters.

▶ **Lemma 8** ([16]). *For node $u \in \mathcal{T}$ outside the subtree of $LCA_{\mathcal{T}}(c_0)$, $L(u)$ is always consistent with c_0.*

Proof. For simplicity, we use lca to denote $LCA_{\mathcal{T}}(c_0)$ here. By the definition of lca, we know $c_0 \subseteq L(lca)$. $u \notin subtree(lca)$ implies that $L(u) \not\subset L(lca)$. Since signature S is consistent, there are two possibilities, either $L(lca) \subset L(u)$ or $L(lca) \cap L(u) = \emptyset$.

- $L(lca) \subset L(u)$: Then $c_0 \subseteq L(lca) \subset L(u)$.
- $L(lca) \cap L(u) = \emptyset$: Then $c_0 \cap L(u) \subseteq L(lca) \cap L(u) = \emptyset$.

In both cases, $L(u)$ is consistent with c_0. ◀

Then we focus on nodes within the subtree of $LCA_{\mathcal{T}}(c_0)$, more specifically, the children of $LCA_{\mathcal{T}}(c_0)$. Following is the characterization.

▶ **Lemma 9** ([16]). *Cluster c_0 is consistent with the signature S of $\mathcal{T}$ if and only if every child w of $LCA_{\mathcal{T}}(c_0)$ satisfies one of the following:*

- $L(w) \subset c_0$
- $L(w) \cap c_0 = \emptyset$

Equivalently, $S \cup \{c_0\}$ is consistent if and only if

$$\sum_{\substack{child\ w\ of\ LCA_{\mathcal{T}}(c_0) \\ L(w) \subset c_0}} |L(w)| = |c_0|$$

Proof. For simplicity, we use lca to denote $LCA_{\mathcal{T}}(c_0)$ here. By Lemma 8, we only have to consider every node u within the subtree of lca. If $c_0 = L(lca)$, c_0 is consistent with S (since $S \cup \{c_0\} = S$), and all children w satisfies $L(w) \subset c_0$. Now we assume $c_0 \neq L(lca)$ which implies $c_0 \subset L(lca)$.

We first prove that this condition is sufficient. For inner node u within the subtree of lca, if $u = lca$, we know $c_0 \subset L(u)$ which means they are consistent. Otherwise, u must belong to the subtree of a child of lca. Let us call this child w_0.

By the condition of this lemma, either $L(w_0) \cap c_0 = \emptyset$ or $L(w_0) \subset c_0$. If $L(w_0) \cap c_0 = \emptyset$, since $L(u) \subseteq L(w_0)$, we know $L(u) \cap c_0 = \emptyset$. Otherwise $L(u) \subseteq L(w_0) \subset c_0$. In either case, $L(u)$ is consistent with c_0.

Then we prove the necessity. For every child w of lca, since $L(w)$ has to be consistent with c_0, either one of two conditions in this lemma holds, or $c_0 \subset L(w)$. If $c_0 \subset L(w)$, then w is either lca or a proper ancestor of lca, which contradicts the fact that w is a child of lca. ◀

3.2 Our Algorithm

Algorithm

We begin with a corollary of the characterization to replace $LCA_{\mathcal{T}}(c_0)$.

▶ **Corollary 10.** *c_0 is consistent with the signature S of $\mathcal{T}$ if and only if, there exists a node $u_0 \in T$, such that $c_0 \subseteq L(u_0)$, and every child w of u_0 satisfies one of the following:*

- $L(w) \subseteq c_0$
- $L(w) \cap c_0 = \emptyset$

Equivalently, $S \cup \{c_0\}$ is consistent if and only if $\exists u_0 \in T$ such that

$$\sum_{\substack{child\ w\ of\ u_0 \\ L(w) \subseteq c_0}} |L(w)| = |c_0|$$

Proof. For necessity, let $u_0 = LCA_{\mathcal{T}}(c)$.

For sufficiency, if $u_0 = LCA_{\mathcal{T}}(c_0)$, it follows from Lemma 9. If $u_0 \neq LCA_{\mathcal{T}}(c_0)$, since $c_0 \subseteq L(u_0)$, there must be a child w_0 of u_0 such that $c_0 \subseteq L(w_0)$. Otherwise, $LCA_{\mathcal{T}}(c_0)$ should have been u_0. On the other hand, because $L(w_0) \cap c_0 \neq \emptyset$, $L(w_0) \subseteq c_0$. Thus $c_0 = L(w_0)$. Namely, if $u_0 \neq LCA_{\mathcal{T}}(c_0)$, it must be the parent of $w_0 = LCA_{\mathcal{T}}(c_0)$, and $c_0 = L(w_0)$. c_0 is then consistent with S since $L(w_0) \in S$. ◀

We have the following lemma to help us find such u_0.

▶ **Lemma 11.** *Suppose x is an arbitrary leaf of $\mathcal{T}$ such that $x \in c_0$. Let u_0 be the lowest ancestor of x that $L(u_0) \nsubseteq c_0$. c_0 is consistent with the signature S of $\mathcal{T}$ if and only if u_0 satisfies the conditions in Corollary 10.*

Besides, let p be the path from the root to u_0. If c_0 is consistent with signature S, for each proper ancestors u of u_0, $L(u) \nsubseteq c_0$, and for each proper descendant u of u_0 on path p, $L(u) \subseteq c_0$.

Proof. If c_0 is consistent with S, by Corollary 10, there is a node $u_0 \in T$ satisfying the conditions. Then by $c_0 \subseteq L(u_0)$, we know x is in the subtree of u_0. In other words, u_0 is on path p.

For each proper ancestor u of u_0, $c_0 \subseteq L(u_0) \subset L(u)$. Thus $L(u) \nsubseteq c_0$. Suppose the child of u_0 on path p is w_0. Since leaf $x \in c_0 \cap L(w_0)$, we know $L(w_0) \subseteq c_0$ from the conditions in Corollary 10. For all proper descendants u of u_0 on path p, $L(u) \subseteq L(w_0) \subseteq c_0$. Thus u_0 is the lowest ancestor of x that $L(u_0) \nsubseteq c_0$. ◀

By Lemma 11, we can perform a binary search on path p to find the lowest ancestor u_0 of x that $L(u_0) \nsubseteq c_0$. We will show how to check whether $L(u) \nsubseteq c_0$ for an arbitrary node $u \in \mathcal{T}$ efficiently in Section 4.

Finally, we discuss how $\mathcal{T}$ should change when we add c_0.

▶ **Lemma 12.** *Suppose S is consistent with c_0 and $c_0 \notin S$. When adding c_0 to S and adding the new node (corresponding to c_0) to $\mathcal{T}$, the u_0 in Lemma 11 should be the parent of the new node on $\mathcal{T}$. The children of the new node should be all children w of u_0 such that $L(w) \subset c_0$.*

Proof. Since the leaf $x \in c_0$ (in Lemma 11), all inner nodes u with $c_0 \subset L(u)$ are on path p, the path from the root to x. Thus by Lemma 11, $L(u_0)$ is the smallest set in S that contains c_0. Then the first claim follows from Observation 2. For those children w of u_0 such that $L(w) \subset c_0$, c_0 becomes the smaller set containing them. Thus they must change their parent. For other nodes u, such that $L(u) \subset c_0$, by the conditions of Corollary 10, u must be within the subtree of a child of u_0. Then that child is a smaller set containing it. Thus their parents stay unchanged. ◀

Algorithm 2 Our algorithm to check consistency and update $\mathcal{T}$.

1: Pick an arbitrary species in c_0, and $x \leftarrow$ the corresponding leaf of $\mathcal{T}$
2: Binary search the path from root to x.
3: $u_0 \leftarrow$ the lowest node u on the path that $L(u) \not\subseteq c_0$
4: $sum \leftarrow \sum_{\text{child } w \text{ of } u_0, L(w) \subseteq c_0} |L(w)|$
5: **if** $sum = |c_0|$ **then**
6: c_0 is consistent with S, and we add it to consensus.
7: We add a new node w' (corresponding to c_0) to $\mathcal{T}$
8: **for** child w of u_0 **do**
9: **if** $L(w) \subset c_0$ **then**
10: Move w to be a child of w'
11: Let w' be a child of u_0

Details of our algorithm are presented in Algorithm 2.

▶ **Lemma 13.** *$sum = |c_0|$ at Line 5, Algorithm 2 if and only if c_0 is consistent with S.*

Proof. If c_0 is consistent with S, by Lemma 11 we must find such a node u_0. On the other hand, if c_0 is not consistent with S, by Corollary 10, there is no such u_0. [1] ◀

To test whether $L(u) \subseteq c_0$ at Line 3, we need the following lemma. Recall $LCA_i(c)$ is the least common ancestor on tree T_i for cluster c.

▶ **Lemma 14.** *Suppose $c_0 = L(v)$ where v is a node in T_i. For a node u in $\mathcal{T}$, $L(u) \subseteq c_0$ if and only if $LCA_i(L(u))$ is in the subtree of v.*

Proof. If $L(u) \subseteq c_0 = L(v)$, every leaf in $L(u)$ is a descendant of v. So $LCA_{T_i}(L(u))$ is in the subtree of v. (note when $c_0 = L(u)$, it is still true since $LCA_i(L(u)) = v$)

Conversely, if $LCA_i(L(u))$ is a descendant of v, since leaves in $L(u)$ are in the subtree of $LCA_i(L(u))$ on T_i, they are also in the subtree of $L(v)$. Thus $L(u) \subseteq L(v) = c_0$. ◀

Thus the subset queries at Line 11 of Algorithm 2 are turned into LCA queries on phylogenetic tree T_i. Also for the summation query at Line 4, $L(w) \subseteq c_0$ if and only if $LCA_i(L(w))$ is in the subtree of v.

Since T_i is a fixed static tree, finding $LCA_i(L(w))$ is tractable in polylogarithmic time. Details are presented in the next section.

[1] Note if c_0 is not consistent with S, u_0 may not equal to $LCA_{\mathcal{T}}(c)$. In this case, we have to refer back to Corollary 10 instead of Lemma 9.

4 Efficiency

In this section, we will show that our algorithm can be implemented efficiently. There are two kinds of queries in Algorithm 2:

1. ***lca*(*u*, *i*)**: Given $u \in \mathcal{T}$, return $LCA_i(L(u))$ on tree T_i. (At Line 3, to see whether $L(u) \subseteq c_0$, by Lemma 14, we need to find out $LCA_i(L(u))$)
2. ***sum*(*u*, *v*, *i*)**: Given $u \in \mathcal{T}$ and $v \in T_i$, evaluate the summation

$$\sum_{\substack{\text{child } w \text{ of } u \\ LCA_i(L(w)) \in subtree(v) \text{ on } T_i}} |L(w)|$$

(By Lemma 14, it equals the summation *sum* at Line 4)

After checking consistency, in Line 7 $\sim$ 11, Algorithm 2, the consensus tree is dynamically updated. The following update operation is needed:

1. ***add*(*u*, *v*, *i*)**: Given $u \in \mathcal{T}$ and $v \in T_i$, add a new node w' to be a child of u. For all children w of u such that $LCA_i(L(w))$ is in the subtree of v, move them to be the children of the new node w'.

4.1 Data Structures

Recall $E(T)$ is the Euler tour of T where we only keep the first occurrence of each node. For node $v \in T_i$, the subtree of $L(v)$ corresponds to a continuous interval $[l_{T_i}(v), r_{T_i}(v)]$ in $E(T_i)$. Let $l_i(v)$ and $r_i(v)$ be the shorthands for $l_{T_i}(v)$ and $r_{T_i}(v)$.

The data structures we use have three components:

1. For each phylogenetic tree T_i, we maintain a top tree T_i'. Thus by Lemma 6, we can answer the least common ancestor of two nodes in $\tilde{O}(1)$ time.
2. For the consensus tree $\mathcal{T}$, we maintain its structure with a top tree $\mathcal{T}'$. By Lemma 6, we can answer k-th ancestor query in $\tilde{O}(1)$ time. This is for the binary search at Line 3, Algorithm 2.
3. For each node $u \in \mathcal{T}$, we use k splay trees $S_1, \ldots, S_k$ to maintain all its children w. The key of child w in the i-th tree is $l_i(LCA_i(L(w)))$, the position of $LCA_i(L(w))$ in $E(T_i)$. We maintain the following two values for each child w:
   - $|L(w)|$: The corresponding operation is integer additions.
   - $LCA_i(L(w))$: The corresponding operation is the least common ancestor of two nodes on T_i.

   Thus the splay trees support the following two kinds of queries:
   - $Sum\_Size(k_1, k_2)$: return the summation of the first value of each w whose key is in range $[k_1, k_2]$.
   - $Sum\_LCA(k_1, k_2)$: return the LCA of the second value of each w whose key is in range $[k_1, k_2]$.

Here the splay trees can be replaced with any balanced search trees with merge and split operations.

For each node $u \in \mathcal{T}$, we compute $LCA_i(L(u))$ for all i once we add u to our consensus tree and store these k numbers at node u. Namely, we compute them during the updates, not the queries.

4.2 Handle Update

Recall $add(u, v, i)$ requires us to do the following:

- add a new node w' to be a child of u on the consensus tree $\mathcal{T}$
- move some children of u to be children of w'

Let the set of all children of u be C and those children need to move be W. $par(u)$ denotes the parent of u.

Here we use the following idea from [16]:

- If $|W| \leq |C|/2$, we add a new node w' to be a child of u. Then we move nodes in W to be the children of w' one by one.
- If $|W| > |C|/2$, instead of moving nodes in W, we move nodes in $C \backslash W$. We disconnect u from $par(u)$ and connect w' with $par(u)$. Namely, replace u with w'. Then we make u a child of w'. For all children of u in $C \backslash W$, we move them to be the children of w'.

In this way, we need to move at most $\min\{|W|, |C|-|W|\}$ nodes. The details are presented in Algorithm 3.

Algorithm 3 Handle update $add(u, v, i)$.

```
1: C ← the set of all children of u
2: W ← {w ∈ C | LCA_i(L(w)) ∈ subtree(v) on T_i}
3: if |W| ≤ |C|/2 then
4:     Add a new node w' to be a child of u on Top tree T'
5:     for w ∈ W do
6:         Cut the edge between w and u on T' and connect w with w'
7:         Remove w from the splay trees at u. Insert w into splay trees at w'
8:     for i ∈ k do
9:         Compute LCA_i(L(w')) ← Sum_LCA(1, n) by query the splay tree S_i at node w'
10: else
11:     Add a new node w' to replace u, and make u a child of w' on Top tree T'
12:     for w ∈ C − W do
13:         Cut the edge between w and u on T' and connect w with w'
14:         Remove w from the splay trees at u. Insert w into splay trees at w'
15:     for i ∈ k do
16:         LCA_i(L(w)) gets the answer we stored at node u before
17:         Compute LCA_i(L(u)) ← Sum_LCA(1, n) by query the splay tree S_i at node u
```

▶ **Lemma 15.** *Each node is moved at most $O(\log n)$ times.*

Proof. No matter we move children in W or $C \backslash W$, the set C is eventually divided into two sets $C \backslash W$ and W after the procedure. Since we always move the nodes in the smaller set, each time we move a node, the size of the set containing it is at least halved. Or equivalently, the number of its siblings is at least halved. Since initially the root has n children, every node is in a set of size n. Each node can be moved at most $O(\log n)$ times. ◀

▶ **Lemma 16.** *All updates $add(u, v, i)$ take $\tilde{O}(kn)$ time in total.*

Proof. See Algorithm 3.

At Line 2, we need to implicitly find out the set W and get its size. By Observation 5, the subtree of v forms a continuous interval in $E(T_i)$. Then the size of W is just the number of nodes with keys within $[l_i(v), r_i(v)]$ in splay tree S_i. We can split this part out from S_i into a splay tree S'. The size of S' is just the size of W.

Then at Line 6, 7, 13, 14, each time we move a node, by Lemma 6 and Lemma 7, it takes $\tilde{O}(k)$ time. (The bottleneck is to delete and insert nodes at all k splay trees) By Lemma 15, each node is moved at most $O(\log n)$ times. Since the consensus tree has at most n nodes in the end, in total this part takes $\tilde{O}(kn)$ time.

At Line 9, and 17, we need to query k splay trees for each node inserted to the consensus tree. By Lemma 7, this takes $\tilde{O}(k)$ time for each node inserted. Thus in total, we need $\tilde{O}(kn)$ time. ◀

4.3 Handle Queries

▶ **Lemma 17.** *The queries $lca(u, i)$ and $sum(u, v, i)$ can be answered in $\tilde{O}(1)$ time.*

Proof. For query $lca(u, i)$, we return the $LCA_i(L(w))$ we computed at Line 17, Algorithm 3 when adding u.

For $sum(u, v, i)$, by Observation 5, all children w that $LCA_i(L(w)) \in subtree(v)$ on T_i are in a continuous interval of $E(T_i)$, namely $[l_i(v), r_i(v)]$. To answer $sum(u, v, i)$, we perform $Sum(l_i(v), r_i(v))$ on splay tree S_i for the second value, namely $|L(w)|$, and we return the summation to be the answer. ◀

4.4 Time complexity

▶ **Theorem 18.** *Greedy consensus tree of k phylogenetic trees for n species can be constructed in $\tilde{O}(kn)$ time.*

Proof. Recall the construction of greedy consensus tree contains two phases:

1. Count the frequency $f(c)$ of clusters and sort them. (Line $1 \sim 3$ of Algorithm 1)
2. Repeatedly run the greedy procedure. (Line $4 \sim 7$ of Algorithm 1)

By Lemma 4, we can get the identifier of each cluster and handle the first phase in $\tilde{O}(kn)$ time.

To handle the second phase, we run our Algorithm 2.

- For the binary search at Line 3, by Lemma 6, we can randomly access the path by k-th ancestor query in $\tilde{O}(1)$ time. By Lemma 14 and Lemma 17, we can check whether $L(u) \subseteq c_0$ in $\tilde{O}(1)$ time.
- For the summation at Line 4, by Lemma 17, can also be evaluated in $\tilde{O}(1)$ time.

Thus for each of the kn clusters, checking consistency takes $\tilde{O}(1)$ time. Then it takes $\tilde{O}(kn)$ time in total.

For Line $7 \sim 11$ of Algorithm 2, we run Algorithm 3. By Lemma 3, this part takes $\tilde{O}(kn)$ time in total.

Thus our algorithm takes $\tilde{O}(kn)$ time. ◀

References

1 Edward N Adams III. Consensus techniques and the comparison of taxonomic trees. *Systematic Biology*, 21(4):390–397, 1972.

2 Alfred V Aho, John E Hopcroft, and Jeffrey D Ullman. On finding lowest common ancestors in trees. *SIAM Journal on computing*, 5(1):115–132, 1976.

3 Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *Acm Transactions on Algorithms (talg)*, 1(2):243–264, 2005.

4 Nina Amenta, Frederick Clarke, and Katherine St John. A linear-time majority tree algorithm. In *International Workshop on Algorithms in Bioinformatics*, pages 216–227. Springer, 2003.

5 Amihood Amir and Dmitry Keselman. Maximum agreement subtree in a set of evolutionary trees: Metrics and efficient algorithms. *SIAM Journal on Computing*, 26(6):1656–1669, 1997.

6 Md Shamsuzzoha Bayzid, Siavash Mirarab, Bastien Boussau, and Tandy Warnow. Weighted statistical binning: enabling statistically consistent genome-scale phylogenetic analyses. *PLoS One*, 10(6):e0129183, 2015.

7 Md Shamsuzzoha Bayzid and Tandy Warnow. Naive binning improves phylogenomic analyses. *Bioinformatics*, 29(18):2277–2284, 2013.

8 Kåre Bremer. Combinable component consensus. *Cladistics*, 6(4):369–372, 1990.

9 David Bryant. A classification of consensus methods for phylogenetics. *DIMACS series in discrete mathematics and theoretical computer science*, 61:163–184, 2003.

10 William HE Day. Optimal algorithms for comparing trees with labeled leaves. *Journal of classification*, 2(1):7–28, 1985.

11 James H Degnan, Michael DeGiorgio, David Bryant, and Noah A Rosenberg. Properties of consensus methods for inferring species trees from gene trees. *Systematic Biology*, 58(1):35–54, 2009.

12 Martin Farach, Teresa M Przytycka, and Mikkel Thorup. Computing the agreement of trees with bounded degrees. In *European Symposium on Algorithms*, pages 381–393. Springer, 1995.

13 Martin Farach and Mikkel Thorup. Optimal evolutionary tree comparison by sparse dynamic programming. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 770–779. IEEE, 1994.

14 James S Farris. On comparing the shapes of taxonomic trees. *Systematic Zoology*, 22(1):50–54, 1973.

15 J Felsenstein. Phylip version 3.6. *Software package, Department of Genome Sciences, University of Washington, Seattle, USA*, 2005.

16 Pawel Gawrychowski, Gad M. Landau, Wing-Kin Sung, and Oren Weimann. A faster construction of greedy consensus trees. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPIcs*, pages 63:1–63:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ICALP.2018.63.

17 Jotun Hein, Tao Jiang, Lusheng Wang, and Kaizhong Zhang. On the complexity of comparing evolutionary trees. In *Annual Symposium on Combinatorial Pattern Matching*, pages 177–190. Springer, 1995.

18 Monika Rauch Henzinger, Valerie King, and Tandy Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24(1):1–13, 1999.

19 Jesper Jansson, Zhaoxian Li, and Wing-Kin Sung. On finding the adams consensus tree. *Information and Computation*, 256:334–347, 2017.

20 Jesper Jansson, Ramesh Rajaby, and Wing-Kin Sung. Minimal phylogenetic supertrees and local consensus trees. *AIMS Medical Science*, 5(2):181, 2018.

21 Jesper Jansson, Chuanqi Shen, and Wing-Kin Sung. Algorithms for the majority rule (+) consensus tree and the frequency difference consensus tree. In *International Workshop on Algorithms in Bioinformatics*, pages 141–155. Springer, 2013.

22 Jesper Jansson, Chuanqi Shen, and Wing-Kin Sung. Improved algorithms for constructing consensus trees. *Journal of the ACM (JACM)*, 63(3):28, 2016.

23 Jesper Jansson, Wing-Kin Sung, Hoa Vu, and Siu-Ming Yiu. Faster algorithms for computing the r* consensus tree. *Algorithmica*, 76(4):1224–1244, 2016.

24 Erich D Jarvis, Siavash Mirarab, Andre J Aberer, Bo Li, Peter Houde, Cai Li, Simon YW Ho, Brant C Faircloth, Benoit Nabholz, Jason T Howard, et al. Whole-genome analyses resolve early branches in the tree of life of modern birds. *Science*, 346(6215):1320–1331, 2014.

25 Sampath Kannan, Tandy Warnow, and Shibu Yooseph. Computing the local consensus of trees. *SIAM Journal on Computing*, 27(6):1695–1724, 1998.

26 Liang Liu, Lili Yu, and Scott V Edwards. A maximum pseudo-likelihood approach for estimating species trees under the coalescent model. *BMC evolutionary biology*, 10(1):302, 2010.

27 Liang Liu, Lili Yu, Laura Kubatko, Dennis K Pearl, and Scott V Edwards. Coalescent methods for estimating phylogenetic trees. *Molecular Phylogenetics and Evolution*, 53(1):320–328, 2009.

28 Timothy Margush and Fred R McMorris. Consensus n-trees. *Bulletin of Mathematical Biology*, 43(2):239–244, 1981.

29 Kurt Mehlhorn, Rajamani Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.

30 Siavash Mirarab, Md Shamsuzzoha Bayzid, and Tandy Warnow. Evaluating summary methods for multilocus species tree estimation in the presence of incomplete lineage sorting. *Systematic Biology*, 65(3):366–380, 2014.

31 James B Pease, David C Haak, Matthew W Hahn, and Leonie C Moyle. Phylogenomics reveals three sources of adaptive variation during a rapid radiation. *PLoS Biology*, 14(2):e1002379, 2016.

32 Fredrik Ronquist and John P Huelsenbeck. Mrbayes 3: Bayesian phylogenetic inference under mixed models. *Bioinformatics*, 19(12):1572–1574, 2003.

33 Leonidas Salichos and Antonis Rokas. Inferring ancient divergences requires genes with strong phylogenetic signals. *Nature*, 497(7449):327, 2013.

34 Leonidas Salichos, Alexandros Stamatakis, and Antonis Rokas. Novel information theory-based measures for quantifying incongruence among phylogenetic trees. *Molecular Biology and Evolution*, 31(5):1261–1271, 2014.

35 Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

36 Jordan V Smith, Edward L Braun, and Rebecca T Kimball. Ratite nonmonophyly: independent evidence from 40 novel loci. *Systematic Biology*, 62(1):35–49, 2012.

37 Alexandros Stamatakis. Raxml version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics*, 30(9):1312–1313, 2014.

38 Alexandros Stamatakis, Paul Hoover, and Jacques Rougemont. A rapid bootstrap algorithm for the raxml web servers. *Systematic biology*, 57(5):758–771, 2008.

39 Wing-Kin Sung. Greedy consensus tree and maximum greedy consensus tree problems. In *International Workshop on Algorithms and Computation*, pages 305–316. Springer, 2019.

40 DL Swofford. Paup*, version 4.0. software package, 2003.