A Faster Construction of Greedy Consensus Trees

Paweł Gawrychowski
Institute of Computer Science, University of Wrocław, Poland
gawry@cs.uni.wroc.pl

Gad M. Landau
University of Haifa, Israel
landau@cs.haifa.ac.il

Wing-Kin Sung
National University of Singapore, Singapore
ksung@comp.nus.edu.sg

Oren Weimann[1]
University of Haifa, Israel
oren@cs.haifa.ac.il

Abstract

A consensus tree is a phylogenetic tree that captures the similarity between a set of conflicting phylogenetic trees. The problem of computing a consensus tree is a major step in phylogenetic tree reconstruction. It is also central for predicting a species tree from a set of gene trees, as indicated recently in [Nature 2013].

This paper focuses on two of the most well-known and widely used consensus tree methods: the greedy consensus tree and the frequency difference consensus tree. Given k conflicting trees each with n leaves, the previous fastest algorithms for these problems were $\mathcal{O}(kn^2)$ for the greedy consensus tree [J. ACM 2016] and $\widetilde{\mathcal{O}}(\min\{kn^2, k^2n\})$ for the frequency difference consensus tree [ACM TCBB 2016]. We improve these running times to $\widetilde{\mathcal{O}}(kn^{1.5})$ and $\widetilde{\mathcal{O}}(kn)$ respectively.

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

Keywords and phrases phylogenetic trees, greedy consensus trees, dynamic trees

Digital Object Identifier 10.4230/LIPIcs.ICALP.2018.63

1 Introduction

A *phylogenetic tree* describes the evolutionary relationships among a set of n species called taxa. It is an unordered rooted tree whose leaves represent the taxa and whose inner nodes represent their common ancestors. Each leaf has a distinct label from $[n]$. The inner nodes are unlabelled and have at least two children.

Numerous phylogenetic trees, reconstructed from data sources like fossils or DNA sequences, have been published in the literature since the early 1860s. However, the phylogenetic trees obtained from different data sources or using different reconstruction methods result in conflicts (similar though not identical phylogenetic trees over the same set $[n]$ of leaf labels). The conflicts between phylogenetic trees are usually measured by their difference in *signatures*: The signature of a phylogenetic tree T is the set $\{\mathsf{L}(u) : u \in T\}$ where $\mathsf{L}(u)$ denotes the set of labels of all leaves in the subtree rooted at node u of T (the set $\mathsf{L}(u)$ is sometimes called a *cluster*). To deal with the conflicts between k phylogenetic trees in a

[1] Supported in part by Israel Science Foundation grant 592/17

45th International Colloquium on Automata, Languages, and Programming (ICALP 2018).
Editors: Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella;
Article No. 63; pp. 63:1–63:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

systematic manner, the concept of a *consensus tree* was invented. Informally, the consensus tree is a single phylogenetic tree that summarizes the branching structure (signatures) of all the conflicting trees. That is, given a collection of k phylogenetic trees with the same set of leaf labels $[n]$, we would like to build a single phylogenetic tree that captures as much of their structure as possible (in practice, we might want to relax this assumption and only require that each set of leaf labels is a subset of $[n]$, but we assume that it is exactly $[n]$ as in the previous theoretical work). Of course, there are many possibilities of how this single phylogenetic tree should be chosen.

Many different types of consensus trees have been proposed in the literature. For almost all of them, optimal or near-optimal $\widetilde{\mathcal{O}}(kn)$ time constructions are known. These include Adam's consensus tree [1], strict consensus tree [27], loose consensus tree [4, 13], majority-rule consensus tree [17, 13], majority-rule (+) consensus tree [11], and asymmetric median consensus tree [20, 21][2]. Two of the most notable exceptions are the frequency difference consensus tree [10] and the greedy consensus tree [5, 9] whose running time remains quadratic in either k or n. In particular, the former can be constructed in $\widetilde{\mathcal{O}}(\min\{kn^2, k^2n\})$ time [11] and the later in $\mathcal{O}(kn^2)$ time [13]. For more details about different consensus trees and their advantages and disadvantages see the survey in [5], Chapter 30 in [8], and Chapter 8.4 in [31].

In this paper we propose novel worst-case efficient algorithms for the frequency difference consensus tree problem and the greedy consensus tree problem.

First, we present an $\mathcal{O}(kn \log^2 n)$ time deterministic labeling method. The labeling method counts the frequency (number of occurrences) of every cluster S in the input trees. Based on this labeling method, we obtain an $\mathcal{O}(kn \log^2 n)$ time construction of the frequency difference consensus tree. Then, for the greedy consensus tree, we present our main technical contribution: a method that uses micro-macro decomposition to verify if a cluster S is compatible with a tree T in $\mathcal{O}(n^{0.5} \log n)$ time and, if so, modify T to include S in $\mathcal{O}(n^{0.5} \log n)$ amortized time. Using this procedure, we obtain an $\mathcal{O}(kn^{1.5} \log n)$ time construction of the greedy consensus tree.

The frequency difference consensus tree. The frequency $f(S)$ of a cluster S (a set of labels of all leaves in some subtree) is the number of trees that contain S. A cluster is said to be *compatible* with another cluster if they are either disjoint or one is included in the other. A *frequent* cluster is a cluster that occurs in more trees than any of the clusters that are incompatible with it. The frequency difference consensus tree is a tree whose signature is exactly all the frequent clusters. Such a tree always exists because, for any pair of incompatible clusters, at most one will be included, and so all included clusters are pairwise compatible.

The frequency difference consensus tree was initially proposed by Goloboff et al. [10], and its relationship with other consensus trees was studied in [7]. In particular, it can be seen as a refinement of the majority-rule consensus tree [17, 13]. Moreover, it is known to give less noisy branches than the greedy consensus tree defined below. Steel and Velasco [30] concluded that "the frequency difference method is worthy of more widespread usage and serious study". A naive construction of the frequency difference consensus tree takes $\mathcal{O}(k^2n^2)$ time. The free software TNT [10] has implemented a heuristics method to construct it more efficiently. However, its time complexity remains unknown.

[2] Constructing the asymmetric median consensus tree was proven to be NP-hard for $k > 2$ [20] and solvable in $\widetilde{\mathcal{O}}(n)$ time for $k = 2$ [21].

Recently, Jansson et al. [11] presented an $\mathcal{O}(\min\{kn^2, k^2n + kn\log^2 n\})$ time construction (implemented in the FACT software package [12]). Their algorithm first computes the frequency $f(S)$ of every cluster S with non-zero frequency. This is done in total $\mathcal{O}(\min\{kn^2, k^2n\})$ time. They then show that given these computed frequencies, the frequency difference consensus tree can be computed in additional $\mathcal{O}(kn\log^2 n)$ time. In Section 2 we show how to compute all frequencies in total $\mathcal{O}(kn\log^2 n)$ time leading to the following theorem:

▶ **Theorem 1.** *The frequency difference consensus tree of k phylogenetic trees $T_1, T_2, \ldots, T_k$ on the same set of leaves $[n]$ can be computed in $\mathcal{O}(kn\log^2 n)$ time.*

To prove the above theorem, we first develop an $\mathcal{O}(kn\log^2 n)$ time algorithm for assigning a number $\mathsf{id}(u) \in [kn]$ to every $u \in T_i$ such that $\mathsf{id}(u) = \mathsf{id}(u')$ iff $\mathsf{L}(u) = \mathsf{L}(u')$. With these numbers in hand, we can then compute the frequencies of all clusters in $\mathcal{O}(kn)$ time using counting sort (since there are only kn clusters with non-zero frequencies, and each was assigned an integer bounded by kn). Notice that this also generates a sorted list of all clusters with non-zero frequencies.

The greedy consensus tree. We say that a given collection $\mathcal{C}$ of subsets of $[n]$ is *consistent* if there exists a phylogenetic tree T such that the signature of T is exactly $\mathcal{C}$. The greedy consensus tree is defined by the following procedure: We begin with an initially empty $\mathcal{C}$ and then consider all clusters S in decreasing order of their frequencies. In this order, for every S, we check if $\mathcal{C} \cup \{S\}$ is consistent, and if so we add S to $\mathcal{C}$.

The greedy consensus tree is one of the most well-known consensus trees. It has been used in numerous papers such as [6, 23, 14, 18, 2, 24, 29, 19, 3, 15, 16, 26, 33] to name a few. For example, in a recent landmark paper in Nature [23], it was used to construct the species tree from 1000 gene trees of yeast genomes, and in [6] it was asserted that *"The greedy consensus tree offers some robustness to gene-tree discordance that may cause other methods to fail to recover the species tree. In addition, the greedy consensus method outperformed our other methods for branch lengths outside the too-greedy zone."*.

The greedy consensus tree is a refinement of the majority-rule consensus tree, and is sometimes called the extended majority-rule consensus (eMRC) tree. It is implemented in popular phylogenetics software packages like PHYLIP [9], PAUP* [32], MrBayes [22], and RAxML [28]. A naive construction of the greedy consensus tree requires $\mathcal{O}(kn^3)$ time [5]. To speed this up, these software packages often use hashing to improve the running time. Thus, if one is interested in analyzing worst-case complexity of the algorithms used in these packages, it would be necessary to allow randomization, as otherwise there is no guarantee on the efficiency of hashing. Even with randomization, the worst-case time complexities of these solutions are not known. Recently, Jansson et al. [13] gave the best known provable construction with an $\mathcal{O}(kn^2)$ deterministic running time (their implementation is also part of the FACT package). In Section 3 we present our main contribution, a deterministic $\widetilde{\mathcal{O}}(kn^{1.5})$ construction as stated by the following theorem:

▶ **Theorem 2.** *The greedy consensus tree of k phylogenetic trees $T_1, T_2, \ldots, T_k$ on the same set of leaves $[n]$ can be computed in $\mathcal{O}(kn^{1.5}\log n)$ time.*

To prove the above theorem, we develop a generic procedure that takes any ordered list of clusters $S_1, S_2, \ldots, S_\ell \subseteq [n]$ and tries adding them one-by-one to the current solution $\mathcal{C}$. We assume that every cluster S_i is specified by providing a tree T_i and a node $u_i \in T_i$ such that $S_i = \mathsf{L}(u_i)$. Our procedure requires $\mathcal{O}(n^{0.5}\log n)$ time per cluster (to add this cluster to $\mathcal{C}$ or assert that it cannot be added) and needs not to assume anything about the order of the clusters. In particular, it does not rely on the clusters being sorted by frequencies.

2 Computing the Identifiers

We process the nodes of every T_i in a bottom-up order. For every node $u \in T_i$, we compute the identifier $\mathsf{id}(u)$ by updating the following structure called the *dynamic set equality structure*:

▶ **Lemma 3** (the dynamic set equality structure)**.** *There exists a data structure that maintains a set of integers under the following operations: (1) create a new empty set in constant time, (2) add $x \in [n]$ to the set in $\mathcal{O}(\log^2 n)$ time, (3) return the identifier of the set in constant time, and (4) list all ℓ elements of the set in $\mathcal{O}(\ell)$ time. The structure ensures that the identifiers are bounded by the total number of update operations performed so far, and that two sets are equal iff their identifiers are equal.*

Proof. To allow for listing all elements of the current set S, we store them in a list. Before adding the new element x to the list, we need to check if $x \in S$. This will be done using the representation described below.

Conceptually, we work with a complete binary tree B on n leaves labelled with $0, 1, \ldots, n-1$ when read from left to right (without losing generality, n is a power of 2), where every node u corresponds to a set $D(u) \subseteq [n]$ defined by the leaves in its subtree (note that $D(u) = \{i, i+1, \ldots, j\}$, where $0 \leq i \leq j < n$). Now, any set S is associated with a binary tree B, where we write 1 in a leaf if the corresponding element belongs to S and 0 otherwise. Then, for every node we define its characteristic vector by writing down the values written in the leaves of its subtree in the natural order (from left to right). Clearly, the vector of an inner node is obtained by concatenating the vector of its children. We want to maintain identifiers of all nodes, so that the identifiers of two nodes are equal iff their characteristic vectors are identical. If we can keep the identifiers small, then the identifier of the current set can be computed as the identifiers of the root of B.

Assume that we have already computed the identifiers of all nodes in B and now want to add x to S. This changes the value in the leaf u corresponding to x and, consequently, the characteristic vectors of all ancestors of u. However, it does not change the characteristic vectors of any other node. Therefore, we traverse the ancestors of u starting from u and recompute their identifiers. Let v be the current node. If we have never seen the characteristic vector of v before, we can set the identifier of v to be the largest already used identifier plus one. Otherwise, we have to set the identifier of v to be the same as the one previously used for a node with such a characteristic vector. As mentioned above, the characteristic vector of an inner node v is the concatenation of the characteristic vectors of its children v_ℓ and v_r. We maintain a dictionary mapping a pair consisting of the identifier of v_ℓ and the identifier of v_r to the identifier of v. The dictionary is global, that is, shared by all instances of the structure. Then, assuming that we have already computed the up-to-date identifiers of v_ℓ and v_r, we only need to query the dictionary to check if the identifier of v should be set to the largest already used identifier plus one (which is exactly when the dictionary does not contain the corresponding pair) or retrieve the appropriate identifier. Therefore, adding x to B reduces to $\log n$ queries to the dictionary. By implementing the dictionary with balanced search trees, we therefore obtain the claimed $\mathcal{O}(\log^2 n)$ time for adding an element.

We are not completely done yet, because creating a new complete binary tree B takes $\mathcal{O}(n)$ time and therefore the initialization time is not constant yet. However, we can observe that it does not make sense to explicitly maintain a node u of B such that $S \cap D(u) = \emptyset$, because we can assume that the identifier of such an u is 0. In other words, we can maintain only the part of B induced by the leaves corresponding to S. Adding an element $x \in S$ is implemented as above, except that we might need to create (at most $\mathcal{O}(\log n)$) new nodes

on the leaf-to-root path corresponding to x (if such a leaf already exists, we terminate the procedure as $x \in S$ already) and then recompute the identifiers on the whole path as described above. ◀

Armed with Lemma 3, we process every T_i bottom-up. Consider an inner node $v \in T_i$ and let $v_1, v_2, \ldots, v_d$ be its children ordered so that $|\mathsf{L}(v_1)| = \max_j |\mathsf{L}(v_j)|$, that is, the subtree rooted at v_1 is the largest. Assuming that we have already stored every $\mathsf{L}(v_j)$ in a dynamic set equality structure, we construct a dynamic set equality structure storing $\mathsf{L}(v)$ by simply inserting all elements of $\mathsf{L}(v_2) \cup \mathsf{L}(v_3) \cup \cdots \cup \mathsf{L}(v_d)$ into the structure of $\mathsf{L}(v_1)$. This takes $\mathcal{O}(\log^2 n)$ time per element. Then, we set $\mathsf{id}(u)$ to be the identifier of the obtained structure. By a standard argument (heavy path decomposition), every leaf of T_i is inserted into at most $\log n$ structures and therefore the whole T_i is processed in $\mathcal{O}(n \log^3 n)$ time. This gives us the claimed $\mathcal{O}(kn \log^3 n)$ total time.

We now proceed with a faster $\mathcal{O}(kn \log^2 n)$ total time solution. While this is irrelevant for our $\mathcal{O}(kn^{1.5} \log n)$ time construction of the greedy consensus tree, it implies a better complexity for constructing the frequency difference consensus tree.

We start with a high-level intuition. Lemma 3 is, in a sense, more than we need, as it is not completely clear that we need to immediately compute the identifier of the current set. Indeed, applying heavy path decomposition we can partially delay computing the identifiers by proceeding in $\mathcal{O}(\log n)$ phases. In each phase, we can then replace the dynamic dictionary used to store the mapping with a radix sort. Intuitively, this shaves one log from the time complexity. We proceed with a detailed explanation.

▶ **Theorem 4.** *The numbers $\mathsf{id}(u)$ can be found for all nodes of the k phylogenetic trees $T_1, T_2, \ldots, T_k$ in $\mathcal{O}(kn \log^2 n)$ total time.*

Proof. For a node $v \in T_i$, define its level $\mathsf{level}(v)$ to be ℓ, such that $2^\ell \leq |\mathsf{L}(v)| < 2^{\ell+1}$. Thus, the levels are between 0 and $\log n$, level of a node is at least as large as the levels of its children, and a node on level ℓ has at most one child on the same level. We work in phases $\ell = 0, 1, \ldots, \log n$. In phase ℓ, we assume that the numbers $\mathsf{id}(v)$ are already known for all nodes v, such that $\mathsf{level}(v) < \ell$, and want to assign these numbers to all nodes v, such that $\mathsf{level}(v) = \ell$. We will show how to achieve this in $\mathcal{O}(kn \log n)$ time, thus proving the theorem.

Consider all nodes v, such that $\mathsf{level}(v) = \ell$. Because every such v has at most one child at the same level, all level-ℓ nodes in T_i can be partitioned into maximal paths of the form $p = v_1 - v_2 - \ldots - v_s$, where the level of the parent of v_1 is larger than ℓ (or v_1 is the root of T_i), and the levels of all children of v_j (except for v_{j+1}, if defined) are smaller than ℓ. v_1 is called the head of p and denoted $\mathsf{head}(p)$. Now, our goal is to find $\mathsf{id}(v_j)$ with the required properties for every $j = 1, 2, \ldots, s$. We will actually achieve a bit more. The sets $\mathsf{L}(\mathsf{head}(p))$ are disjoint in every tree T_i, and thus we can define, for every i, a partition $\mathcal{P}_i = \{P_i(1), P_i(2), \ldots, P_i(t_i)\}$ of the set of leaves $[n]$, where every $P_i(z)$ corresponds to a level-ℓ path $p = v_1 - v_2 - \ldots - v_s$ in T_i, such that $\mathsf{L}(\mathsf{head}(p)) = P_i(z)$. The elements of $P_i(z)$ are then ordered, and we think that $P_i(z)$ is a sequence of length $|P_i(z)|$. The ordering is chosen so that, for every $j = 1, 2, \ldots, s$, the set $\mathsf{L}(v_j)$ corresponds to some prefix of $P_i(z)$. $P_i(z)[1..r]$ denotes the prefix of $P_i(z)$ of length r. We will assign identifiers to all such prefixes $P_i(z)[1..r]$, for every $i = 1, 2, \ldots, k$, $z = 1, 2, \ldots, t_i$ and $r = 1, 2, \ldots, |P_i(z)|$, with the property that the identifiers of two prefixes are equal iff the sets of leaves appearing in both of them are equal. Then, we can extract the required $\mathsf{id}(v_j)$ in constant time each by taking the identifiers of some $P_i(z)[1..r]$.

Recall that in the slower solution we worked with a complete binary tree B on n leaves. For every set S in the collection and every $u \in B$, we computed an identifier of the set $S \cap D(u)$. This was possible, because if u_ℓ and u_r are the left and the right child of u,

respectively, then the identifier of $S \cap D(u)$ can be found using the identifiers of $S \cap D(u_\ell)$ and $S \cap D(u_r)$. We need to show that retrieving these identifiers can be batched.

Fix a node $u \in B$ and, for every $i = 1, 2, \ldots, k$ and $z = 1, 2, \ldots, t_i$, consider all prefixes $P_i(z)[1..r]$ for $r = 1, 2, \ldots, |P_i(z)|$. We create a *version* of u for every such prefix. The version corresponds to the set containing all elements of $D(u)$ occurring in the prefix $P_i(z)[1..r]$. We want to assign identifiers to all versions of u. First, observe that we only have to create a new version if $P_i(z)[r] \in D(u)$, as otherwise the set is the same as for $r-1$. Thus, the total number of required versions, when summed over all nodes $u \in B$ on the same depth in B, is only kn, as a leaf of T_i creates exactly one new version for some u. For every node $u \in B$, we will store a list of all its versions. A version consists of its identifier (such that the identifier of two versions is the same iff the corresponding sets are equal) together with the indices i, z and r. We describe how to create such a list for every node $u \in B$ at the same depth d given the lists for all nodes at depth $d+1$ next.

Let u_1 and u_2 be the left and the right child of $u \in B$, respectively. Then, we need to create a new version of u for every new version of u_1 and every new version of u_2, because for the set corresponding to u to change either the set corresponding to u_1 or the set corresponding to u_2 must change, and every change is adding one new element. Fix i and z and consider all versions of u_1 corresponding to i and z sorted according to r. Let the sorted list of their r's be $a_1 < a_2 < \ldots$. Similarly, consider all versions of u_2 corresponding to i and z sorted according to r, and let the sorted list of their r's be $b_1 < b_2 < \ldots$. For every $x \in \{a_1, a_2, \ldots\} \cup \{b_1, b_2, \ldots\}$, we create a new version of u corresponding to i, z, and r equal to x. This is done by retrieving the version of u_1 with r equal to a_p, such that $a_p \leq x$ and p is maximized, and the version of u_2 with r equal to b_q, such that $b_q \leq x$ and q is maximized. Then, the identifier of the new version of u can be constructed from the pair consisting of the identifiers of these versions of u_1 and u_2 (this is essentially the same reasoning as in the slower solution). We could now use a dictionary to map these pairs to identifiers. However, we can also observe that, in fact, we have reduced finding the identifiers of all versions of all nodes $u \in B$ at the same depth d to identifying duplicates on a list of kn pairs of numbers from $[kn]$. This can be done by radix sorting all pairs in linear time (more precisely, $\mathcal{O}(kn)$ time and $\mathcal{O}(kn)$ space), and then sweeping through the sorted list while assigning the identifiers. This takes only $\mathcal{O}(kn)$ time for every depth d, so $\mathcal{O}(kn \log n)$ for every level as claimed. ◀

The proof of Theorem 1 follows immediately from Theorem 4.

3 Simulating the Greedy Algorithm

We consider k trees $T_1, \ldots, T_k$ on the same set of leaves $[n]$, and assume that every node u has an identifier $\mathsf{id}(u)$ such that $\mathsf{id}(u) = \mathsf{id}(u')$ iff $\mathsf{L}(u) = \mathsf{L}(u')$. We next develop a general method for maintaining a solution $\mathcal{C}$ (i.e., a set of compatible identifiers) so that, given any node $u \in T_i$, we are able to efficiently check if $\mathsf{L}(u)$ is compatible with $\mathcal{C}$, meaning that $\mathcal{C} \cup \mathsf{L}(u)$ is consistent, and if so add $\mathsf{L}(u)$ to $\mathcal{C}$. Our method does not rely on the order in which the sets arrive and in particular can be used to run the greedy algorithm.

We represent $\mathcal{C}$ with a phylogenetic tree T_c such that $\mathcal{C} = \{\mathsf{L}(u) : u \in T_c\}$. T_c is called the *current consensus tree*. By Lemma 2.2 of [13], S is compatible with $\mathcal{C}$ iff the node $v \in T_c$ defined as the lowest common ancestor of all leaves with labels from S has the property that, for every child v' of v, either $\mathsf{L}(v') \cap S = \emptyset$ or $\mathsf{L}(v') \subseteq S$. Recall that the lowest common ancestor (lca) of u and v is the deepest node w that is an ancestor of both u and v. Also, adding $\mathsf{L}(u)$ to $\mathcal{C}$ can be done by creating a new child w of v and reconnecting every original child v' of v such that $\mathsf{L}(v') \subseteq S$ to the new w. This is illustrated in Figure 1 (left).

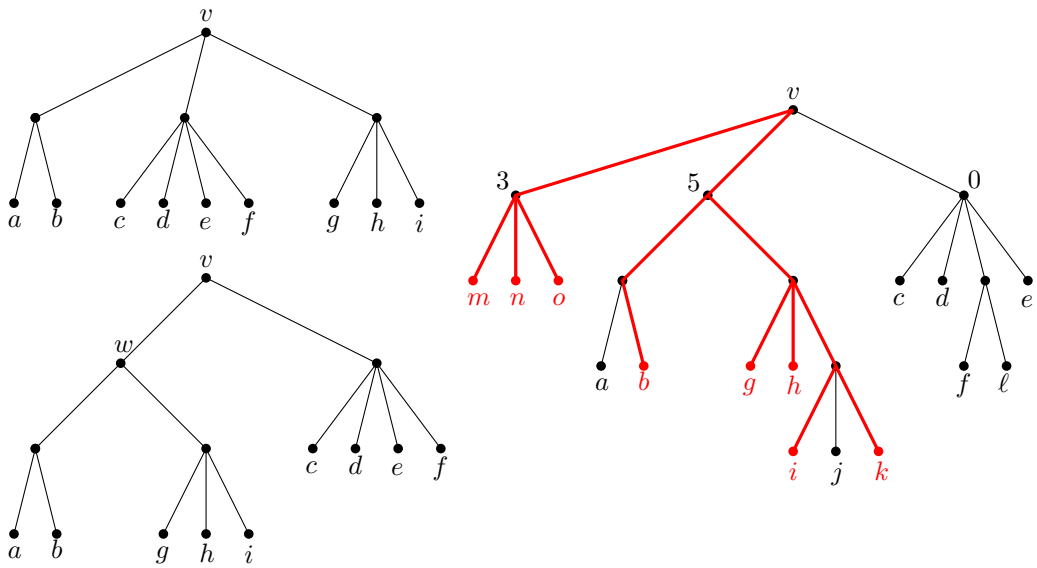

Figure 1 Left: adding $\{a, b, g, h, i\}$ to S. Right: checking if $S = \{m, n, o, b, g, h, i, k\}$ is compatible with T_c. Leaves corresponding to the elements of S are shown in red and their lca is v. S is not compatible with T_c because the counter of the middle child of v is equal to 5 yet there are 7 leaves in its subtree.

Initially, T_c consists only of n leaves attached to the common root (which corresponds to $\mathcal{C} = \{\{x\} : x \in [n]\}$). Our goal is to maintain some additional information so that given any node $u \in T_i$, we can check if $\mathsf{L}(u)$ is compatible with $\mathcal{C}$ in $\mathcal{O}(n^{0.5} \log n)$ time. After adding $\mathsf{L}(u)$ to $\mathcal{C}$ the information will be updated in amortized $\mathcal{O}(kn^{0.5} \log n)$ time. To explain the intuition, we first show how to check if $\mathsf{L}(u)$ is compatible with $\mathcal{C}$ in roughly $\mathcal{O}(|\mathsf{L}(u)|)$ time.

Let $\mathsf{L}(u) = \{\ell_1, \ell_2, \ldots, \ell_s\}$ and let u_i be the leaf of T_c labelled with ℓ_i. Let v be the lowest common ancestor of $u_1, u_2, \ldots, u_s$ found by asking $s - 1$ lca queries: we start with u_1 and then iteratively jump to the lca of the current node and u_i. Assuming that we represent T_c in such a way that an lca query can be answered efficiently, this takes roughly $\mathcal{O}(s)$ time. Then, we need to decide if for every child v' of v it holds that $\mathsf{L}(v') \subseteq \mathsf{L}(u)$ or $\mathsf{L}(v') \cap \mathsf{L}(u) = \emptyset$. This can be done by computing, for every such v', how many u_i's belong to the subtree rooted at v', and then checking if this number is either 0 or $|\mathsf{L}(v')|$. To compute these numbers, we maintain a counter for every v'. Then, for every u_i we retrieve the child v' of v such that u_i belongs to the subtree rooted at v' and increase the counter of v'. Assuming that we represent T_c so that such v' can be retrieved efficiently, this again takes roughly $\mathcal{O}(s)$ time. Finally, we iterate over all u_i again, retrieve the corresponding v' and check if its counter is equal to $|\mathsf{L}(v')|$ (so our representation of T_c should also allow retrieving the number of leaves in a subtree). If not, then $\mathsf{L}(u)$ is not compatible with $\mathcal{C}$, see Figure 1 (right). Otherwise, we create the new node w and reconnect to w all children v' of v, such that the counter of v' is equal to $|\mathsf{L}(v')|$.

We would like to avoid explicitly iterating over all elements of $\mathsf{L}(u)$. This will be done by maintaining some additional information, so that we only have to iterate over up to $n^{0.5}$ elements. To explain what is the additional information we need the (standard) notion of a *micro-macro decomposition*. Let b be a parameter and consider a binary tree on n nodes. We want to partition it into $\mathcal{O}(n/b)$ node-disjoint subtrees called *micro trees*. Each micro tree is of size at most b and contains at most two *boundary nodes* that are adjacent to nodes

in other micro trees. One of these boundary nodes, called the top boundary node, is the root of the whole micro tree, and the other is called the bottom boundary node. Such a partition is always possible and can be found in $\mathcal{O}(n)$ time.

We binarize every T_i to obtain T_i' (this could be avoided by working with edge-disjoint subtrees in the decomposition, but we find node-disjoint subtrees easier to think about; binarization adds a number of artificial nodes u for which we do not check if $\mathsf{L}(u)$ is compatible with $\mathcal{C}$)). Then, we find a micro-macro decomposition of T_i' with $b = n^{0.5}$ (where b has been chosen as to minimize the total running time). By properties of the decomposition we have the following:

▶ **Proposition 5.** *For any $u \in T_i$ such that $|\mathsf{L}(u)| > n^{0.5}$, there exists a boundary node $v \in T_i'$ such that $\mathsf{L}(u)$ can be obtained by adding at most $n^{0.5}$ elements to $\mathsf{L}(v)$. Furthermore, v and these up to $n^{0.5}$ elements can be retrieved in $\mathcal{O}(n^{0.5})$ time after $\mathcal{O}(n)$ preprocessing.*

The total number of boundary nodes is only $\mathcal{O}(kn^{0.5})$. For each such boundary node u, we maintain a pointer to a node $\mathsf{finger}(u) \in T_c$ called the *finger* of u, defined as the lowest common ancestor in T_c of all leaves with labels belonging to $\mathsf{L}(u)$. Additionally, the children of $\mathsf{finger}(u)$ are partitioned into three groups: (1) v_i such that $\mathsf{L}(v_i) \subseteq \mathsf{L}(u)$, (2) v_i such that $\mathsf{L}(v_i) \cap \mathsf{L}(u) = \emptyset$, and (3) the rest. We call them full, empty, and mixed, respectively (with respect to u). For each group we maintain a list storing all nodes in the group, every node knows its group, and the group knows it size. Additionally, every group knows the total number of leaves in all subtrees rooted at its nodes.

We also need to augment the representation T_c to allow for efficient *extended lca queries*. An extended lca query, denoted $\mathsf{lca\_ext}(u, v)$, returns the first edge on the path from the lca of u and v to u, and -1 if u is an ancestor of v. For example, in Figure 1 (right), $\mathsf{lca\_ext}(v, k) = -1$ whereas $\mathsf{lca\_ext}(h, k)$ is the edge between v and its leftmost child. The following lemma follows by slightly tweaking the link/cut trees of Sleator and Tarjan [25].

▶ **Lemma 6.** *We can maintain a collection of rooted trees under: (1) create a new tree consisting of a single node, (2) make the root of one tree a child of a node in another tree, (3) delete an edge from a node to its parent, (4) count leaves in the tree containing a given node, and (5) extended lca queries, all in $\mathcal{O}(\log n)$ amortized time, where n is the total size of all trees in the collection.*

We next show how to efficiently check for any u if $\mathsf{L}(u)$ is compatible with $\mathcal{C}$. By the following lemma, this can be done in $\mathcal{O}(n^{0.5} \log n)$ time, assuming we have stored the required additional information. Recall that this additional information includes:

1. The tree T_c maintained using Lemma 6.
2. For every boundary node w, we store $\mathsf{finger}(w)$.
3. For every boundary node w, we store three lists containing the full, the mixed, and the empty children of $\mathsf{finger}(w)$ respectively. Each list also stores the total number of leaves in all subtrees rooted at its nodes.

▶ **Lemma 7.** *Assuming access to the above additional information, given any node $u \in T_i$ we can check if $\mathsf{L}(u)$ is compatible with $\mathcal{C}$ in $\mathcal{O}(n^{0.5} \log n)$ time.*

Proof. By Lemma 2.2 of [13], to check if $\mathsf{L}(u)$ is compatible with $\mathcal{C}$ we need to check if, for a node v defined as the lowest common ancestor of all leaves with labels belonging to $\mathsf{L}(u)$, it holds that for every child v' of v either $\mathsf{L}(v') \cap \mathsf{L}(u) = \emptyset$ or $\mathsf{L}(v') \subseteq \mathsf{L}(u)$. By properties of the micro-macro decomposition, we can retrieve a boundary node w and a set S of up to $n^{0.5}$ labels such that $\mathsf{L}(u) = \mathsf{L}(w) \cup S$ (if $|\mathsf{L}(u)| < n^{0.5}$, there is no w). See Figure 2. Then, the

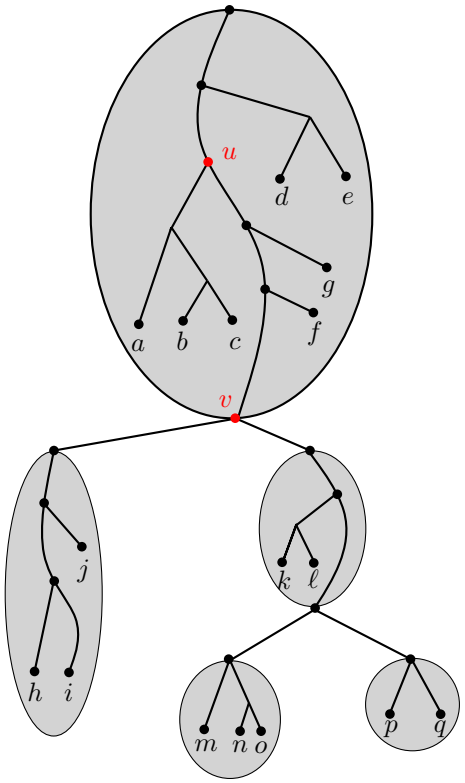

Figure 2 A schematic illustration of the micro-macro decomposition. v is a boundary node and $\mathsf{L}(v) = \{h, i, j, k, \ell, m, n, o, p, q\}$. Then, $\mathsf{L}(u) = \{a, b, c, f, g, h, i, j, k, \ell, m, n, o, p, q\}$ so $\mathsf{L}(u) = \mathsf{L}(v) \cup \{a, b, c, f, g\}$.

lowest common ancestor of all leaves with labels belonging to $\mathsf{L}(u)$ is the lowest common ancestor of $\mathsf{finger}(w)$ and all leaves with labels belonging to S. Therefore, v can be found with $|S|$ lca queries in $\mathcal{O}(n^{0.5} \log n)$ time. Second, to check if $\mathsf{L}(v_i) \cap \mathsf{L}(u) = \emptyset$ or $\mathsf{L}(v_i) \subseteq \mathsf{L}(u)$ for every child v_i of v we distinguish two cases:

(1) If v is a proper ancestor of $\mathsf{finger}(w)$ we can calculate $|\mathsf{L}(v_i) \cap \mathsf{L}(u)|$ for every v_i in $\mathcal{O}(|S| \log n) = \mathcal{O}(n^{0.5} \log n)$ time as follows. Every edge has its associated counter. We assume that all counters are set to zero before starting the procedure and will make sure that they are cleared at the end. First, we use an $\mathsf{lca\_ext}(w, v)$ query to access the edge leading to the subtree containing w and set its counter to $|\mathsf{L}(w)|$. Then, we iterate over all $\ell \in S$, retrieve the leaf u of T_c labelled with ℓ, and use an $\mathsf{lca\_ext}(u, v)$ query to access the edge leading to the subtree of v containing u and increase its counter by one. Additionally, whenever we access an edge for the first time (in this particular query), we add it to a temporary list Q. After having processed all $\ell \in S$, we iterate over $(v, v_i) \in Q$ and check if the counter of (v, v_i) is equal to the number of leaves in the subtree rooted at v_i (which requires retrieving the number of leaves). If this condition holds for every $(v, v_i) \in Q$ then $\mathsf{L}(u)$ is compatible with $\mathcal{C}$ and furthermore, the nodes v_i such that $(v, v_i) \in Q$ are exactly the ones that should be reconnected. Finally, we iterate over the edges in Q again and reset their counters.

(2) If $v = \mathsf{finger}(w)$ the situation is a bit more complicated because we might not have enough time to explicitly iterate over all children of v that should be reconnected. Nevertheless, we can use a very similar method. Every edge has its associated counter (again,

we assume that the counter are set to zero before starting the procedure and will make sure that they are cleared at the end). We also need a global counter g, that is set to the total number of leaves in all subtrees rooted at either full or mixed children of v decreased by $|\mathsf{L}(w)|$. g can be initialized in constant time in the first step of the procedure due to the additional information stored with every list of children. Intuitively, g is how many leaves not belonging to $\mathsf{L}(w)$ we still have to see to conclude that indeed $\mathsf{L}(v_i) \cap \mathsf{L}(u) = \emptyset$ or $\mathsf{L}(v_i) \subseteq \mathsf{L}(u)$ for every child v_i of v. We iterate over $\ell \in S$ and access the edge (v, v_i) leading to the subtree containing u labelled with ℓ. We decrease g by one and, if v_i is an empty child of v and this is the first time we have seen v_i (in this query) then we add the number of leaves in the subtree rooted at v_i to g. If, after having processed all $\ell \in S$, $g = 0$ then we conclude that $\mathsf{L}(u)$ is compatible with $\mathcal{C}$. The whole process takes $\mathcal{O}(|S| \log n) = \mathcal{O}(n^{0.5} \log n)$ time. ◀

Before explaining the details of how to update the additional information, we present the intuition. Recall that adding $\mathsf{L}(u)$ to $\mathcal{C}$ is done by creating a new child v' of v and reconnecting some children of v to v'. Let the set of all children of v be C and the set of children that should be reconnected be C_r. Note that if $|C_r| = 1$ or $|C| = |C_r|$ then we do not have to change anything in T_c. Otherwise, updating T_c can be implemented using two different methods:

1. Delete edges from nodes in C_r to v. Create a new tree consisting of a single node v' and make it a child of v. Then, make all nodes in C_r children of v'.
2. Delete edges from nodes in $C \setminus C_r$ to v. Delete the edge from v to its parent w. Create a new tree consisting of a single node v' and make it a child of w. Then, make v a child of v' and also make all nodes in $C \setminus C_r$ children of w. See Figure 3.

Thus, by using C_r or $C \setminus C_r$, the number of operations can be either $\mathcal{O}(|C_r|)$ or $\mathcal{O}(|C| - |C_r|)$. We claim that by choosing the cheaper option we can guarantee that the total time for modifying the link-cut tree representation of T_c is $\mathcal{O}(n \log^2 n)$. Intuitively, every edge of the final consensus tree participates in $\mathcal{O}(\log n)$ operations, and there are at most n such edges. This is formalized in the following lemma.

▶ **Lemma 8.** $\min\{|C_r|, |C| - |C_r|\}$ *summed over all updates of* T_c *is* $n \log n$.

Proof. We assume that $2 \leq |C_r| < |C|$ in every update, as otherwise there is nothing to change in T_c. Then, there are at most n updates, as each of them creates a new inner node and there are never any nodes with degree 1 in T_c.

We bound the sum of $\min\{|C_r|, |C| - |C_r|\}$ by assigning credits to inner nodes of T_c. During the execution of the algorithm, a node u with b siblings should have $\log b$ credits. Thus, whenever we create a new inner node we need at most $\log n$ new credits, thus the total number of allocated credits is $n \log n$. It remains to argue that, whenever we create a new child v' of v and reconnect some of its children, the original credits of v can be used to pay for the update and make sure that all children of v and v' have enough credits after the update.

Denoting $x = |C_r|$ and $y = |C| - |C_r|$, the cost of the update is $\min\{x, y\}$. The total number of credits of all children of v before the update is $(x + y) \log(x + y - 1)$. After the update, the number of credits of all children of v is $(y + 1) \log y \leq y \log y + \log n$ and the number of credits of all children of v' is $x \log(x - 1)$. Ignoring the $\log n$ new credits allocated to v', the number of available credits is thus:

$$(x + y) \log(x + y - 1) - y \log y - x \log(x - 1) = x \log(1 + y/(x - 1)) + y \log(1 + (x - 1)/y)$$

which is at least $\min\{x, y\}$ for $x \geq 2$, so enough to pay $\min\{|C_r|, |C| - |C_r|\}$ for the update. Hence, the sum is at most $n \log n$. ◀

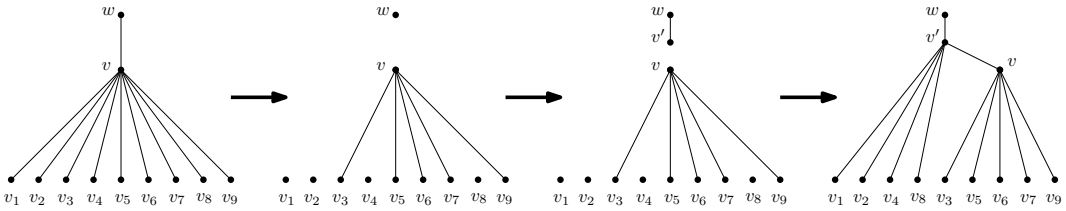

Figure 3 Reconnecting children v_3, v_5, v_6, v_7, v_9 of v using the second method.

Before presenting the whole update procedure, we need one more technical lemma.

▶ **Lemma 9.** *The procedure for checking if* $\mathsf{L}(u)$ *is compatible with* $\mathcal{C}$ *can be requested to return* C_r *in* $\mathcal{O}(|C_r| + n^{0.5})$ *time or* $C \setminus C_r$ *in* $\mathcal{O}(|C| - |C_r| + n^{0.5})$ *time.*

Proof. By inspecting the proof of Lemma 7, we see that there are two cases depending on whether v is a proper ancestor of $\mathsf{finger}(w)$ or not.

1. If v is a proper ancestor of $\mathsf{finger}(w)$ then C_r can be obtained from Q. More precisely, for every $(v, v_i) \in Q$ we add v_i to C_r in $\mathcal{O}(|C_r|)$ total time. We can also obtain $C \setminus C_r$ in $\mathcal{O}(|C|) = \mathcal{O}(|C \setminus C_r| + |S|) = \mathcal{O}(|C| - |C_r| + n^{0.5})$ time.
2. If $v = \mathsf{finger}(w)$ then, while iterating over $\ell \in S$, if this is the first time we have seen v_i then we add v_i to C_r. Additionally, we add all full children of v to C_r. Thus, C_r can be generated in $\mathcal{O}(|C_r|)$ time. Similarly, $C \setminus C_r$ consists of all empty children of v without the nodes v_i seen when iterating over $\ell \in S$, and so can be generated in $\mathcal{O}(|C \setminus C_r| + |S|) = \mathcal{O}(|C| - |C_r| + n^{0.5})$ time.

Thus, we can always generate C_r in $\mathcal{O}(|C_r| + n^{0.5})$ time and $C \setminus C_r$ in $\mathcal{O}(|C| - |C_r| + n^{0.5})$ time. ◀

To add $\mathsf{L}(u)$ to $\mathcal{C}$, we will need to iterate over either C_r or $C \setminus C_r$ (depending on which is smaller). After paying additional $\mathcal{O}(n^{0.5})$ time we can assume that we have access to a list of the elements in the appropriate set. The additional time sums up to $\mathcal{O}(n^{1.5})$, because there can be only n distinct new sets added to $\mathcal{C}$.

▶ **Lemma 10.** *If* $\mathsf{L}(u)$ *is compatible with* $\mathcal{C}$ *then, after adding* $\mathsf{L}(u)$ *to* $\mathcal{C}$ *and modifying* T_c *we can update all additional information in amortized* $\mathcal{O}(kn^{0.5} \log n)$ *time assuming that we add* n *such sets.*

Proof. Recall that T_c is maintained using the data structure from Lemma 6, and adding $\mathsf{L}(u)$ to $\mathcal{C}$ is implemented by creating a new child v' of v and reconnecting some of the children of v to v'. C is the set of all children of v and C_r is the set of children of v that are reconnected to v'. If $|C_r| \leq |C| - |C_r|$ we iterate over C_r and reconnect them one-by-one. If $|C_r| > |C| - |C_r|$ we iterate over $C \setminus C_r$ and reconnect them to a new node w that is inserted between v and its parent. To iterate over either C_r or $C \setminus C_r$, we extend the query procedure as explained in Lemma 9. This adds $\mathcal{O}(n^{0.5})$ to the time complexity, but then we can assume that the requested set can be generated in time proportional to its size. To unify the case of $|C_r| \leq |C| - |C_r|$ and $|C_r| > |C| - |C_r|$, we think that v is replaced with two nodes v' and v'', where v' is the parent of v''. All nodes in C_r become children of v'' while all nodes of $C \setminus C_r$ become children of v' after iterating over either C_r or $C \setminus C_r$, depending on which set is smaller, so by Lemma 8 in the whole process we iterate over sets of total size $n \log n$, so only amortized $\log n$ assuming that we add n sets $\mathsf{L}(u)$.

Consider a boundary node u. If $\mathsf{finger}(u) \neq v$ then there is no need to update the additional information concerning u. If $\mathsf{finger}(u) = v$ then we need to decide if the finger of u should be set to v' or v'' and update the partition of the children of $\mathsf{finger}(u)$ accordingly.

$\mathsf{finger}(u)$ should be set to v' exactly when, for any $w \in C \setminus C_r$, $\mathsf{L}(w) \cap \mathsf{L}(u) = \emptyset$ or, in other words, all nodes in $C \setminus C_r$ are empty with respect to u. The groups should be updated as follows:

1. If $\mathsf{finger}(u)$ is set to v'' then we should remove all nodes in $C \setminus C_r$ from the list of empty nodes with respect to u (as they are no longer children of $\mathsf{finger}(u)$). Other groups remain unchanged.
2. If $\mathsf{finger}(u)$ is set to v' then we should remove all nodes in C_r from the lists. Additionally, we need to insert v'' into the appropriate group: full if all nodes in C_r were full, empty if all nodes in C_r were empty, and mixed otherwise.

We need to show that all these conditions can be checked by either iterating over the nodes of C or over the nodes of $C \setminus C_r$, because we want to iterate over the smaller of these. This then guarantees that the amortized cost of updating the additional information for a boundary node is only $\mathcal{O}(\log n)$, so amortized $\mathcal{O}(kn^{0.5} \log n)$ overall.

To check if all nodes in $C \setminus C_r$ are empty with respect to u, we can either iterate over the nodes in $C \setminus C_r$ or iterate over all nodes in C_r and check if all nodes in C that are full or empty in fact belong to C_r (this is possible because we also keep the total number of full and empty nodes in C). Thus, we can check if $\mathsf{finger}(u)$ should be set to v'.

If $\mathsf{finger}(u)$ is set to v' we need to decide where to put v''. We only explain how to decide if all nodes in C_r are full, as the procedure for empty is symmetric. We can either iterate over all nodes in C_r and check that they are full or iterate over all nodes in $C \setminus C_r$ and check that all nodes in C that are empty or mixed in fact belong to $C \setminus C_r$ (and thus do not belong to C_r, so all nodes in C_r are full). Finally, we add the number of leaves in the subtree rooted at v'' (extracted in $\mathcal{O}(\log n)$ time) to the appropriate sum.

It remains to describe how to remove all unnecessary nodes from the lists. Here we do not worry about having to iterate over the smaller set, because there are only $\mathcal{O}(n)$ new edges created during the whole execution of the algorithm, so we can afford to explicitly iterate over the nodes that should be removed, that is, over C or $C \setminus C_r$. For every removed node, we also subtract the number of leaves in its subtree (extracted in $\mathcal{O}(\log n)$ time) from the appropriate sum. Overall, this adds $\mathcal{O}(n \log n)$ per boundary node to the time complexity, so only amortized $\mathcal{O}(kn^{0.5} \log n)$ overall. ◀

References

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

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

3 M.S. Bayzid, S. Mirarab, B. Boussau, and T. Warnow. Weighted statistical binning: enabling statistically consistent genome-scale phylogenetic analyses. *PLOS One*, page e0129183, 2015.

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

5 D. Bryant. A classification of consensus methods for phylogenetics. In *Bioconsensuss*, volume 61 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 163–184. American Mathematical Society, 2003.

6 J. H. Degnan, M. DeGiorgio, D. Bryant, and N. A. Rosenberg. Properties of consensus methods for inferring species trees from gene trees. *Systematic Biology*, 58(1):35–54, 2009.

7 J. Dong, D. Fernández-Baca, F. R. McMorris, and R. C. Powers. Majority-rule (+) consensus trees. *Mathematical Biosciences*, 228(1):10–15, 2010.

8 J. Felsenstein. *Inferring Phylogenies*. Sinauer Associates, Inc., Sunderland, Massachusetts, 2004.

9 J. Felsenstein. PHYLIP, version 3.6. Software package, Department of Genome Sciences, University of Washington, Seattle, U.S.A., 2005.

10 P. A. Goloboff, J. S. Farris, and K. C. Nixon. TNT, a free program for phylogenetic analysis. *Cladistics*, 24(5):774–786, 2008.

11 Jesper Jansson, Ramesh Rajaby, Chuanqi Shen, and Wing-Kin Sung. Algorithms for the majority rule (+) consensus tree and the frequency difference consensus tree. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2016.

12 Jesper Jansson, Chuanqi Shen, and Wing-Kin Sung. Fast Algorithms for Consensus Trees (FACT). `http://compbio.ddns.comp.nus.edu.sg/~consensus.tree`, 2013.

13 Jesper Jansson, Chuanqi Shen, and Wing-Kin Sung. Improved algorithms for constructing consensus trees. *Journal of the ACM*, 63(3):1–24, 2016.

14 E. D. Jarvis et al. Whole-genome analyses resolve early branches in the tree of life of modern birds. *Science*, 346(6215):1320–1331, 2014.

15 Liang Liu, Lili Yu, and Scott Edwards. A maximum pseudo-likelihood approach for estimating species trees under the coalescent model. *BMC Evolutionary Biology*, 10:302, 2010.

16 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.

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

18 S. Mirarab, S. Bayzid, and T. Warnow. Evaluating summary methods for multilocus species tree estimation in the presence of incomplete lineage sorting. *Systematic Biology*, 65(3):366–380, 2016.

19 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):1–24, 2016.

20 Cynthia Phillips and Tandy J. Warnow. The asymmetric median tree — a new model for building consensus trees. *Discrete Applied Mathematics*, 71(1-3):311–335, 1996.

21 Ramesh Rajaby and Wing-Kin Sung. Computing asymmetric median tree of two trees via better bipartite matching algorithm. In *IWOCA*, 2017.

22 F. Ronquist and J. P. Huelsenbeck. MrBayes 3: Bayesian phylogenetic inference under mixed models. *Bioinformatics*, 19(12):1572–1574, 2003.

23 L. Salichos and A. Rokas. Inferring ancient divergences requires genes with strong phylogenetic signals. *Nature*, 497:327–331, 2013.

24 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.

25 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

26 Jordan V. Smith, Edward L. Braun, and Rebecca T. Kimball. Ratite nonmonophyly: independent evidence from 40 novel loci. *Systematic Biology*, 62(1):35–49, 2013.

27 R. R. Sokal and F. J. Rohlf. Taxonomic congruence in the Leptopodomorpha re-examined. *Systematic Zoology*, 30(3):309–325, 1981.

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

29 Alexandros Stamatakis, Paul Hoover, Jacques Rougemont, and Susanne Renner. A rapid bootstrap algorithm for the raxml web servers. *Systematic Biology*, 57(5):758, 2008.

30 M. Steel and J. D. Velasco. Axiomatic opportunities and obstacles for inferring a species tree from gene trees. *Systematic Biology*, 63(5):772–778, 2014.

31 Wing-Kin Sung. *Algorithms in Bioinformatics: A Practical Introduction*. Chapman & Hall/CRC, Boca Raton, Florida, 2010.

32 D. L. Swofford. PAUP*, version 4.0. Software package, Sinauer Associates, Inc., Sunderland, Massachusetts, 2003.

33 Jimmy Yang and Tandy J. Warnow. Fast and accurate methods for phylogenomic analyses. *BMC Bioinformatics*, 12(S-9):S4, 2011.