Algorithms for the Majority Rule (+) Consensus Tree and the Frequency Difference Consensus Tree

Jesper Jansson, Ramesh Rajaby, Chuanqi Shen, and Wing-Kin Sung

Abstract—This article presents two new deterministic algorithms for constructing consensus trees. Given an input of k phylogenetic trees with identical leaf label sets and n leaves each, the first algorithm constructs the *majority rule (+) consensus tree* in $O(kn)$ time, which is optimal since the input size is $\Omega(kn)$, and the second one constructs the *frequency difference consensus tree* in $\min\{O(kn^2), O(kn(k + \log^2 n))\}$ time.

Index Terms—Phylogenetic tree, consensus tree, cluster, pairwise compatibility, tree algorithm

1 Introduction

A *consensus tree* is a phylogenetic tree that summarizes a given collection of phylogenetic trees having the same leaf labels but different branching structures. Consensus trees are used to resolve structural differences between two or more existing phylogenetic trees arising from conflicts in the raw data, to find strongly supported groupings, and to reconcile large sets of candidate trees obtained by bootstrapping when trying to infer a new phylogenetic tree accurately [1], [2], [3], [4].

Since the first type of consensus tree was proposed by Adams [5] in 1972, many others have been defined and analyzed. See, e.g., [6], Chapter 30 in [3], or Chapter 8.4 in [4] for some surveys. Which particular type of consensus tree to use in practice depends on the context. For example, the *strict consensus tree* [7] is very intuitive and easy to compute [8] and may be sufficient when there is not so much disagreement in the data. On the other hand, the *majority rule consensus tree* [9] is "the optimal tree to report if we view the cost of reporting an estimate of the phylogeny to be a linear function of the number of incorrect clades in the estimate and the number of true clades that are missing from the estimate and we view the reporting of an incorrect grouping as a more serious error than missing a clade" [10]. As another example, the *R* consensus tree* [6] may be useful when combining gene trees [2]. Therefore, scientists need algorithms for constructing a broad range of different consensus trees.

In a recent series of articles [11], [12], [13], [14], [15], we have developed fast algorithms for computing the *majority rule consensus tree* [9], the *loose consensus tree* [16] (also known in the literature as the *combinable component consensus tree* or the *semi-strict consensus tree*), a *greedy consensus tree* [6], [17], the *Adams consensus tree* [5], the *R* consensus tree* [6], and consensus trees for so-called *multi-labeled phylogenetic trees (MUL-trees)* [18]. In this article, we study two relatively new types of consensus trees called the *majority rule (+) consensus tree* [19], [20] and the *frequency difference consensus tree* [21], and give algorithms for constructing them efficiently.

1.1 Definitions and Notation

We shall use the following basic definitions. A *phylogenetic tree* is a rooted, unordered, leaf-labeled tree in which every internal node has at least two children and all leaves have different labels. (Below, phylogenetic trees are referred to as "trees" for short). For any tree T, the set of all nodes in T is denoted by $V(T)$ and the set of all leaf labels in T by $\Lambda(T)$. Any nonempty subset C of $\Lambda(T)$ is called a *c luster* of $\Lambda(T)$; if $|C| = 1$ or $C = \Lambda(T)$ then C is *trivial*, and otherwise, C is *non-trivial*. For any $u \in V(T)$, $T[u]$ denotes the subtree of T rooted at the node u. Observe that if u is the root of T or if u is a leaf then $\Lambda(T[u])$ is a trivial cluster. The set $\mathcal{C}(T) = \bigcup_{u \in V(T)} \{\Lambda(T[u])\}$ is called the *cluster collection of* T, and any cluster $C \subseteq \Lambda(T)$ is said to *occur in* T if $C \in \mathcal{C}(T)$.

Two clusters $C_1, C_2 \subseteq \Lambda(T)$ are *compatible* if $C_1 \subseteq C_2$, $C_2 \subseteq C_1$, or $C_1 \cap C_2 = \emptyset$. If C_1 and C_2 are compatible, we write $C_1 \smile C_2$; otherwise, $C_1 \not\smile C_2$. A cluster $C \subseteq \Lambda(T)$ is *compatible with* T if $C \smile \Lambda(T[u])$ holds for every node $u \in V(T)$. In this case, we write $C \smile T$, and $C \not\smile T$ otherwise. If T_1 and T_2 are two trees with $\Lambda(T_1) = \Lambda(T_2)$ such that every cluster in $\mathcal{C}(T_1)$ is compatible with T_2 then it follows that every cluster in $\mathcal{C}(T_2)$ is compatible with T_1, and we say

- *J. Jansson is with the Laboratory of Mathematical Bioinformatics (Akutsu Laboratory), Institute for Chemical Research, Kyoto University, Gokasho Uji, Kyoto 611-0011, Japan. E-mail: jj@kuicr.kyoto-u.ac.jp.*
- *R. Rajaby is with the NUS Graduate School for Integrative Sciences and Engineering, National University of Singapore, 28 Medical Drive, Singapore 117456. E-mail: e0011356@u.nus.edu, ramesh.rajaby@gmail.com.*
- *C. Shen is with Stanford University, 450 Serra Mall, Stanford, CA 94305-2004. E-mail: shencq@stanford.edu.*
- *W.-K. Sung is with the School of Computing, National University of Singapore, 13 Computing Drive, Singapore 117417, and the Genome Institute of Singapore, 60 Biopolis Street, Genome, Singapore 138672. E-mail: ksung@comp.nus.edu.sg.*

Manuscript received 25 May 2016; revised 17 Aug. 2016; accepted 2 Sept. 2016. Date of publication 20 Sept. 2016; date of current version 2 Feb. 2018. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TCBB.2016.2609923

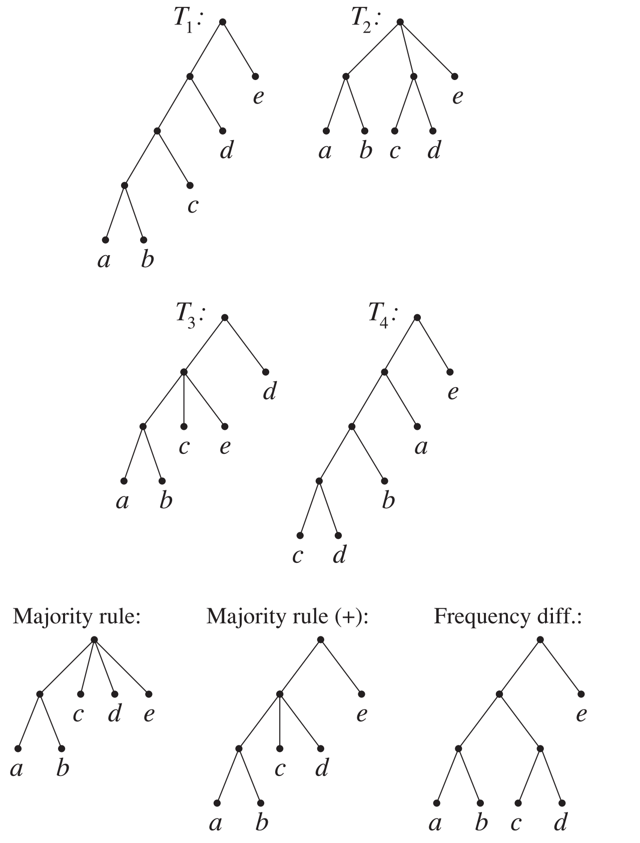


Fig. 1. Let $\mathcal{S} = \{T_1, T_2, T_3, T_4\}$ as shown above with $L = \Lambda(T_1) = \Lambda(T_2) = \Lambda(T_3) = \Lambda(T_4) = \{a, b, c, d, e\}$. The only non-trivial majority cluster of $\mathcal{S}$ is $\{a, b\}$, the non-trivial majority (+) clusters of $\mathcal{S}$ are $\{a, b\}$ and $\{a, b, c, d\}$, and the non-trivial frequency difference clusters of $\mathcal{S}$ are $\{a, b\}$, $\{a, b, c, d\}$, and $\{c, d\}$. The majority rule, majority rule (+), and frequency difference consensus trees of $\mathcal{S}$ are displayed.

that T_1 and T_2 are *compatible*. Any two clusters or trees that are not compatible are called *incompatible*.

Next, let $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$ be a set of trees satisfying $\Lambda(T_1) = \Lambda(T_2) = \ldots = \Lambda(T_k) = L$ for some leaf label set L. For any cluster C of L, denote the set of all trees in $\mathcal{S}$ in which C occurs by $K_C(\mathcal{S})$ and the set of all trees in $\mathcal{S}$ that are incompatible with C by $Q_C(\mathcal{S})$. Thus, $K_C(\mathcal{S}) = \{T_i : C \in \mathcal{C}(T_i)\}$ and $Q_C(\mathcal{S}) = \{T_i : C \not\hookrightarrow T_i\}$. Define three special types of clusters:

- If $|K_C(\mathcal{S})| > \frac{k}{2}$ then C is a *majority cluster of* $\mathcal{S}$.
- If $|K_C(\mathcal{S})| > |Q_C(\mathcal{S})|$ then C is a *majority (+) cluster of* $\mathcal{S}$.
- If $|K_C(\mathcal{S})| > \max\{|K_D(\mathcal{S})| : D \subseteq L \text{ and } C \not\hookrightarrow D\}$ then C is a *frequency difference cluster of* $\mathcal{S}$.

(In other words, a frequency difference cluster is a cluster that occurs more frequently than each of the clusters that is incompatible with it.) According to the definitions, a majority cluster of $\mathcal{S}$ is always a majority (+) cluster of $\mathcal{S}$ and a majority (+) cluster of $\mathcal{S}$ is always a frequency difference cluster of $\mathcal{S}$, but not the other way around, as illustrated in Fig. 1.

The *majority rule consensus tree of* $\mathcal{S}$ [9] is the tree T such that $\Lambda(T) = L$ and $\mathcal{C}(T)$ consists of all majority clusters of $\mathcal{S}$. Similarly, the *majority rule (+) consensus tree of* $\mathcal{S}$ [19], [20] is the tree T such that $\Lambda(T) = L$ and $\mathcal{C}(T)$ consists of all majority (+) clusters of $\mathcal{S}$, and the *frequency difference consensus tree of* $\mathcal{S}$ [21] is the tree T such that $\Lambda(T) = L$ and $\mathcal{C}(T)$ consists of all frequency difference clusters of $\mathcal{S}$. See Fig. 1 for some examples. By Theorem 1 in [9], Lemma 2 in [20], and Proposition 3 in [22], respectively, each of these three consensus trees always exists and is uniquely defined.

From here on, $\mathcal{S}$ is assumed to be an input set of trees with identical leaf label sets. The leaf label set of the trees in $\mathcal{S}$ is denoted by L. To express the size of the input, we define $k = |\mathcal{S}|$ and $n = |L|$.

1.2 Previous Work

Margush and McMorris [9] introduced the majority rule consensus tree in 1981, and a deterministic algorithm for constructing it in optimal $O(kn)$ worst-case running time was presented in [13]. (A randomized algorithm with $O(kn)$ expected running time and unbounded worst-case running time was given earlier by Amenta et al. [1].) The majority rule consensus tree has several desirable mathematical properties [10], [23], [24], and algorithms for constructing it have been implemented in popular computational phylogenetics packages like PHYLIP [17], TNT [25], COMPONENT [26], MrBayes [27], SumTrees in DendroPy [28], and PAUP* [29]. Consequently, it is one of the most widely used consensus trees in practice [19, p. 450].

One drawback of the majority rule consensus tree is that it may be too harsh and discard valuable branching information. For example, in Fig. 1, even though the cluster $\{a, b, c, d\}$ is compatible with 75 percent of the input trees, it is not included in the majority rule consensus tree. For this reason, people have become interested in alternative types of consensus trees that include all the majority clusters and at the same time, also include other meaningful, well-defined kinds of clusters. The majority rule (+) consensus tree and the frequency difference consensus tree are two such consensus trees.

The majority rule (+) consensus tree was defined by Dong et al. [20] in 2010. It was obtained as a special case of an attempted generalization by Cotton and Wilkinson [19] of the majority rule consensus tree. According to [20], Cotton and Wilkinson [19] suggested two types of supertrees[1] called majority-rule (-) and majority-rule (+) that were supposed to generalize the majority rule consensus tree. Unexpectedly, only the first one did, and by restricting the second one to the consensus tree case, [20] arrived at the majority rule (+) consensus tree. Dong et al. [20] established some fundamental properties of this type of consensus tree and pointed out the existence of a polynomial-time algorithm for constructing it, but left the task of finding the best possible such algorithm as an open problem. As far as we know, no implementation for computing the majority rule (+) consensus tree is publicly available.

Goloboff et al. [21] initially proposed the frequency difference consensus tree as a way to improve methods for evaluating group support in parsimony analysis. Its relationships to other consensus trees have been investigated in [20]. In [22], Steel and Velasco studied a generalization of the frequency difference consensus tree to the supertree setting and concluded that "the frequency-difference method is worthy of more widespread usage and serious study". A method for constructing the frequency difference consensus tree has been implemented in the free software package

1. A *supertree* is a generalization of a consensus tree that does not require the input trees to have identical leaf label sets.

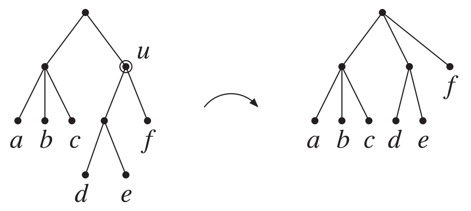


Fig. 2. Figure from [13]. Applying a $delete$ operation on u removes the cluster $\{d,e,f\}$ from the cluster collection. Symmetrically, one can insert the cluster $\{d,e,f\}$ into the cluster collection of the rightmost tree by performing an $insert$ operation with v in the definition above equal to the root and $X = \{x, f\}$, where x is the parent of d and e, which results in the leftmost tree.

TNT [25] but the algorithm used is not documented and its time complexity is unknown. We note that since the number of clusters occurring in $\mathcal{S}$ may be $\Omega(kn)$, a naive algorithm that compares every cluster in $\mathcal{S}$ to every other cluster in $\mathcal{S}$ directly would require $\Omega(k^2n^2)$ time.

1.3 Organization of the Article and New Results

The article is organized as follows. Section 2 summarizes some results from the literature that are needed later. In Section 3, we modify the techniques from [13] to obtain an $O(kn)$-time algorithm for the majority rule (+) consensus tree. Its running time is optimal because the size of the input is $\Omega(kn)$; hence, we resolve the open problem of Dong et al. [20] mentioned above. Next, Section 4 gives a $\min\{O(kn^2), O(kn(k+\log^2 n))\}$-time algorithm for constructing the frequency difference consensus tree (here, the second term is smaller than the first term if $k = o(n)$; e.g., if $k = O(1)$ then the running time reduces to $O(n\log^2 n)$). Both of the new algorithms are deterministic. Finally, Section 5 presents prototype implementations (which are not fully deterministic) of our algorithms and experimental results, and Section 6 gives some concluding remarks.

2 Preliminaries

2.1 The $delete$ and $insert$ operations

The $delete$ and $insert$ operations are two operations that modify the structure of a tree. They are defined in the following way.

Let T be a tree and let u be any non-root, internal node in T. The $delete$ operation on u makes all of u's children become children of the parent of u, and then removes u and the edge between u and its parent. See Fig. 2. The time needed for this operation is proportional to the number of children of u, and the effect of applying it is that the cluster collection of T is changed to $\mathcal{C}(T) \setminus \{\Lambda(T[u])\}$.

Conversely, for any specified existing internal node v in a tree T and any proper subset X of v's children satisfying $|X| \geq 2$, the $insert$ operation first removes all edges between v and X and then creates a new internal node u that is set to be: (1) a child of v; and (2) the parent of all nodes in X. The effect is that $\mathcal{C}(T)$ is changed to $\mathcal{C}(T) \cup \{\Lambda(T[u])\}$, where $\Lambda(T[u]) = \bigcup_{v_i \in X} \Lambda(T[v_i])$.

2.2 Subroutines

The new algorithms in this article use the following algorithms from the literature as subroutines: Day's algorithm [8], Procedure `One-Way_Compatible` [13], and Procedure `Merge_Trees` [13]. Day's algorithm [8] is used to efficiently check whether any specified cluster that occurs in a tree T also occurs in another tree T_{ref}, and can be applied to find the set of all clusters that occur in both T and T_{ref} in linear time. Procedure `One-Way_Compatible` takes as input two trees T_A and T_B with identical leaf label sets and outputs a copy of T_A in which every cluster that is not compatible with T_B has been removed. (The procedure is asymmetric; e.g., if T_A consists of n leaves attached to a root node and $T_B \neq T_A$ then `One-Way_Compatible`$(T_A, T_B) = T_A$, while `One-Way_Compatible`$(T_B, T_A) = T_B$.) Procedure `Merge_Trees` takes as input two compatible trees with identical leaf label sets and outputs a tree that combines their cluster collections. Their properties are summarized below; for details, see references [8] and [13].

Lemma 1 (Day [8]). *Let T_{ref} and T be two given trees with $\Lambda(T_{ref}) = \Lambda(T) = L$ and let $n = |L|$. After $O(n)$ time preprocessing, it is possible to determine, for any $u \in V(T)$, if $\Lambda(T[u]) \in \mathcal{C}(T_{ref})$ in $O(1)$ time.*

Lemma 2 ([13]). *Let T_A and T_B be two given trees with $\Lambda(T_A) = \Lambda(T_B) = L$ and let $n = |L|$. Procedure `One-Way_Compatible`(T_A, T_B) returns a tree T with $\Lambda(T) = L$ such that $\mathcal{C}(T) = \{C \in \mathcal{C}(T_A) : C$ is compatible with $T_B\}$ in $O(n)$ time.*

Lemma 3 ([13]). *Let T_A and T_B be two given trees with $\Lambda(T_A) = \Lambda(T_B) = L$ that are compatible and let $n = |L|$. Procedure `Merge_Trees`(T_A, T_B) returns a tree T with $\Lambda(T) = L$ and $\mathcal{C}(T) = \mathcal{C}(T_A) \cup \mathcal{C}(T_B)$ in $O(n)$ time.*

3 Constructing the Majority Rule (+) Consensus Tree

This section presents an algorithm named `Maj_Rule_Plus` for computing the majority rule (+) consensus tree of $\mathcal{S}$ in (optimal) $O(kn)$ time.

The pseudocode of `Maj_Rule_Plus` is given in Fig. 3. The algorithm has two phases. Phase 1 examines the input trees, one by one, to construct a set of candidate clusters that includes all majority (+) clusters. Then, Phase 2 removes all candidate clusters that are not majority (+) clusters.[2]

During Phase 1, the current candidate clusters are stored as nodes in a tree T. Every node v in T represents a current candidate cluster $\Lambda(T[v])$ and has a counter $count(v)$ that, starting from the iteration at which $\Lambda(T[v])$ became a candidate cluster, keeps track of the number of input trees in which it occurs minus the number of input trees that are incompatible with it. More precisely, while treating the tree T_j for any $j \in \{2, 3, \ldots, k\}$ in Step 3.1, $count(v)$ for each current candidate cluster $\Lambda(T[v])$ is updated as follows: if $\Lambda(T[v])$ occurs in T_j then $count(v)$ is incremented by 1, if $\Lambda(T[v])$ does not occur in T_j and is not compatible with T_j then $count(v)$ is decremented by 1, and otherwise (i.e., $\Lambda(T[v])$ does not occur in T_j but is compatible with T_j)

2. This basic strategy was previously used in the $O(kn)$-time algorithm in [13] for computing the majority rule consensus tree. The main difference is how the counters are updated in Phase 1; instead of producing a superset of the majority clusters as in [13], we now produce a superset of the majority (+) clusters.

```
Algorithm Maj_Rule_Plus
Input: A set $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$ of trees with $\Lambda(T_1) = \Lambda(T_2) = \ldots = \Lambda(T_k)$.
Output: The majority rule (+) consensus tree of $\mathcal{S}$.
  /* Phase 1 */
1 $T := T_1$
2 for each $v \in V(T)$ do $count(v) := 1$
3 for $j := 2$ to $k$ do
3.1     for each $v \in V(T)$ do
            if $\Lambda(T[v])$ occurs in $T_j$ then
                $count(v) := count(v) + 1$
            else if $\Lambda(T[v])$ is not compatible with $T_j$ then
                $count(v) := count(v) - 1$
        endfor
3.2     for each $v \in V(T)$ in top-down order do
            if $count(v) = 0$ then delete node $v$
3.3     for every $C \in \mathcal{C}(T_j)$ that is compatible with $T$ but
        does not occur in $T$ do
            insert $C$ into $T$ and initialize $count(v) := 1$ for
            the new node $v$ satisfying $\Lambda(T[v]) = C$
        endfor
  endfor
  /* Phase 2 */
4 for each $v \in V(T)$ do $K(v) := 0$; $Q(v) := 0$
5 for $j := 1$ to $k$ do
5.1     for each $v \in V(T)$ do
            if $\Lambda(T[v])$ occurs in $T_j$ then
                $K(v) := K(v) + 1$
            else if $\Lambda(T[v])$ is not compatible with $T_j$ then
                $Q(v) := Q(v) + 1$
        endfor
6 for each $v \in V(T)$ in top-down order do
        if $K(v) \leq Q(v)$ then delete node $v$
7 return $T$
End Maj_Rule_Plus
```

Fig. 3. Algorithm `Maj_Rule_Plus` for constructing the majority rule (+) consensus tree.

$count(v)$ is unchanged. Furthermore, if any $count(v)$ reaches 0 then the node v is deleted from T so that $\Lambda(T[v])$ is no longer a current candidate cluster. Next, in Step 3.3, every cluster occurring in T_j that is not a current candidate but compatible with T is inserted into T (thus becoming a current candidate cluster) and its counter is initialized to 1. Lemma 4 below proves that the set of majority (+) clusters of $\mathcal{S}$ is contained in the set of candidate clusters at the end of Phase 1.

Lemma 4. *For any $C \subseteq L$, if C is a majority (+) cluster of $\mathcal{S}$ then $C \in \mathcal{C}(T)$ at the end of Phase 1.*

Proof. Suppose that C is a majority (+) cluster of $\mathcal{S}$. Let T_x be any tree in $Q_C(\mathcal{S})$ and consider iteration x in Step 3: If C is a current candidate at the beginning of iteration x then its counter will be decremented, cancelling out the occurrence of C in one tree T_j where $1 \leq j < x$; otherwise, C may be prevented from being inserted into T in at most one later iteration j (where $x < j \leq k$ and $C \in \mathcal{C}(T_j)$) because of some cluster occurring in T_x. It follows from $|K_C(\mathcal{S})| - |Q_C(\mathcal{S})| > 0$ that C's counter will be greater than 0 at the end of Phase 1, and therefore $C \in \mathcal{C}(T)$. □

In Phase 2, Step 5 of the algorithm computes the values of $|K_C(\mathcal{S})|$ and $|Q_C(\mathcal{S})|$ for every candidate cluster C and stores them in $K(v)$ and $Q(v)$, respectively, where $C = \Lambda(T[v])$. Finally, Step 6 removes every candidate cluster C that does not satisfy the condition $|K_C(\mathcal{S})| > |Q_C(\mathcal{S})|$. By definition, the clusters that remain in T are the majority (+) clusters.

Theorem 1. *Algorithm `Maj_Rule_Plus` constructs the majority rule (+) consensus tree of $\mathcal{S}$ in $O(kn)$ time.*

Proof. The correctness follows from Lemma 4 and the above discussion.

The time complexity analysis is analogous to the proof of Theorem 3.3 in [13]. First consider Phase 1. Step 3.1 takes $O(n)$ time by: (1) running Day's algorithm with $T_{ref} = T_j$ and then checking each node v in $V(T)$ to see if $\Lambda(T[v])$ occurs in T_j (according to Lemma 1, this requires $O(n)$ time for preprocessing, and each of the $O(n)$ nodes in $V(T)$ may be checked in $O(1)$ time), and (2) computing $X :=$ `One-Way_Compatible`(T, T_j) and then checking for each node v in $V(T)$ if v does not exist in X to determine if $\Lambda(T[v]) \not\leftarrow T_j$ (this takes $O(n)$ time by Lemma 2). The *delete* operations in Step 3.2 take $O(n)$ time because the nodes are handled in top-down order, which means that for every node, its parent will change at most once in each iteration. In Step 3.3, define $Y :=$ `One-Way_Compatible` (T_j, T) and $Z :=$ `Merge_Trees`(Y, T). Then by Lemmas 2 and 3, the cluster collection of Y consists of the clusters occurring in T_j that are compatible with the set of current candidates, and Z is the result of inserting these clusters into T. Thus, Step 3.3 can be implemented by computing Y and Z, updating T's structure according to Z, and setting the counters of all new nodes to 1, so Step 3.3 takes $O(n)$ time. The main loop in Step 3 consists of $O(k)$ iterations, and Phase 1 therefore takes $O(kn)$ time in total.

Next, Phase 2 also takes $O(kn)$ time because Step 5.1 can be implemented in $O(n)$ time with the same techniques as in Step 3.1, and Step 6 is performed in $O(n)$ time by handling the nodes in top-down order so that each node's parent is changed at most once, as in Step 3.2. □

4 Constructing the Frequency Difference Consensus Tree

Here, we present an algorithm for finding the frequency consensus tree of $\mathcal{S}$ in $\min\{O(kn^2), O(kn(k + \log^2 n))\}$ time. It is called `Frequency_Difference` and is described in Section 4.1. The algorithm uses the procedure `Merge_Trees` as well as a new procedure named `Filter_Clusters` whose details are given in Section 4.2.

For each tree $T_j \in \mathcal{S}$ and each node $u \in V(T_j)$, define the *weight of* u as the value $|K_{\Lambda(T_j[u])}(\mathcal{S})|$, i.e., the number of trees from $\mathcal{S}$ in which the cluster $\Lambda(T_j[u])$ occurs, and denote it by $w(u)$. For convenience, also define $w(C) = w(u)$, where $C = \Lambda(T_j[u])$. The input to Procedure `Filter_Clusters` is two trees T_A, T_B with $\Lambda(T_A) = \Lambda(T_B) = L$ such that every cluster occurring in T_A or T_B also occurs in at least one tree in $\mathcal{S}$, and the output is a copy of T_A in which every cluster that is incompatible with some cluster in T_B with a higher weight has been removed. Formally, the output

```
Algorithm  Frequency_Difference
Input:  A set $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$ of trees with $\Lambda(T_1) = \Lambda(T_2) = \ldots = \Lambda(T_k)$.
Output: The frequency difference consensus tree of $\mathcal{S}$.
    /* Preprocessing */
1   Compute $w(C)$ for every cluster $C$ occurring in $\mathcal{S}$.
    /* Main algorithm */
2   $T := T_1$
3   for $j := 2$ to $k$ do
        $A :=$ Filter_Clusters$(T, T_j)$
        $B :=$ Filter_Clusters$(T_j, T)$
        $T :=$ Merge_Trees$(A, B)$
    endfor
4   for $j := 1$ to $k$ do
        $T :=$ Filter_Clusters$(T, T_j)$
5   return $T$
End Frequency_Difference
```

Fig. 4. Algorithm `Frequency_Difference` for constructing the frequency difference consensus tree.

of `Filter_Clusters` is a tree T with $\Lambda(T) = L$ such that $\mathcal{C}(T) = \{\Lambda(T_A[u]) : u \in V(T_A)$ and $w(u) > w(x)$ for every $\in V(T_B)$ with $\Lambda(T_A[u]) \not\sim \Lambda(T_B[x])\}$.

4.1 Algorithm `Frequency_Difference`

We first describe Algorithm `Frequency_Difference`. Refer to Fig. 4 for the pseudocode.

The algorithm starts by computing the weight $w(C)$ of every cluster C occurring in $\mathcal{S}$ in a preprocessing step (Step 1). Next, let $\mathcal{C}(S)$ for any set S of trees denote the union $\bigcup_{T_i \in S} \mathcal{C}(T_i)$, and for any $j \in \{1, 2, \ldots, k\}$, define a *forward frequency difference consensus tree of* $\{T_1, T_2, \ldots, T_j\}$ as any tree that includes every cluster C in $\mathcal{C}(\{T_1, T_2, \ldots, T_j\})$ satisfying $w(C) > w(X)$ for all $X \in \mathcal{C}(\{T_1, T_2, \ldots, T_j\})$ with $C \not\sim X$. Steps 2–3 use Procedure `Filter_Clusters` from Section 4.2 to build a tree T that, after any iteration $j \in \{1, 2, \ldots, k\}$, is a forward frequency difference consensus tree of $\{T_1, T_2, \ldots, T_j\}$, as proved in Lemma 5 below. After iteration k, $\mathcal{C}(T)$ contains all frequency difference clusters of $\mathcal{S}$ but possibly some other clusters as well, so Step 4 applies `Filter_Clusters` again to remove all non-frequency difference clusters of $\mathcal{S}$ from T.

Lemma 5. *For any $j \in \{2, 3, \ldots, k\}$, suppose that T is a forward frequency difference consensus tree of $\{T_1, T_2, \ldots, T_{j-1}\}$. Let $A :=$ `Filter_Clusters`(T, T_j) and $B :=$ `Filter_Clusters`(T_j, T). Then `Merge_Trees`(A, B) is a forward frequency difference consensus tree of $\{T_1, T_2, \ldots, T_j\}$.*

Proof. We first show that A and B are compatible. For the sake of obtaining a contradiction, suppose that A and B are not compatible. This means there exist two clusters $C_A \in \mathcal{C}(A)$ and $C_B \in \mathcal{C}(B)$ such that $C_A \not\sim C_B$. However, $A :=$ `Filter_Clusters`(T, T_j) means that $w(C_A) > w(X)$ for all $X \in \mathcal{C}(T_j)$ with $C_A \not\sim X$, and in particular, $w(C_A) > w(C_B)$. Analogously, $B :=$ `Filter_Clusters`(T_j, T) means that $w(C_B) > w(C_A)$, which yields a contradiction. We conclude that A and B are compatible.

Next, consider the result of computing `Merge_Trees`(A, B). By definition, $\mathcal{C}(T)$ includes every cluster C in $\mathcal{C}(\{T_1, T_2, \ldots, T_{j-1}\})$ satisfying $w(C) > w(X)$ for all $X \in \mathcal{C}(\{T_1, T_2, \ldots, T_{j-1}\})$ with $C \not\sim X$. Observe that:

(1) Since $A :=$ `Filter_Clusters`(T, T_j), $\mathcal{C}(A)$ is a subset of $\mathcal{C}(T)$ that includes every cluster C in $\mathcal{C}(\{T_1, T_2, \ldots, T_{j-1}\})$ satisfying $w(C) > w(X)$ for all $X \in \mathcal{C}(\{T_1, T_2, \ldots, T_j\})$ with $C \not\sim X$.

(2) Similarly, $\mathcal{C}(B)$ includes every cluster C in $\mathcal{C}(T_j)$ satisfying $w(C) > w(X)$ for all $X \in \mathcal{C}(\{T_1, T_2, \ldots, T_{j-1}\})$ with $C \not\sim X$. It follows immediately that $\mathcal{C}(B)$ includes every cluster C in $\mathcal{C}(T_j)$ satisfying $w(C) > w(X)$ for all $X \in \mathcal{C}(\{T_1, T_2, \ldots, T_j\})$ with $C \not\sim X$.

By (1) and (2), $\mathcal{C}(A) \cup \mathcal{C}(B)$ contains every cluster C in $\mathcal{C}(\{T_1, T_2, \ldots, T_j\})$ satisfying $w(C) > w(X)$ for all $X \in \mathcal{C}(\{T_1, T_2, \ldots, T_j\})$ with $C \not\sim X$. This shows that `Merge_Trees`(A, B) is a forward frequency difference consensus tree of $\{T_1, T_2, \ldots, T_j\}$. □

Theorem 2. *Algorithm `Frequency_Difference` constructs the frequency difference consensus tree of $\mathcal{S}$ in $\min\{O(kn^2), O(k^2 n)\} + O(k \cdot f(n))$ time, where $f(n)$ is the running time of Procedure `Filter_Clusters`.*

Proof. After completing iteration k of Step 3, $\mathcal{C}(T)$ is a superset of the set of all frequency difference clusters of $\mathcal{S}$ by Lemma 5. Next, Step 4 removes all non-frequency difference clusters of $\mathcal{S}$, so the output will be the frequency difference consensus tree of $\mathcal{S}$.

To analyze the time complexity, first consider how to compute all the weights in Step 1. One method is to first fix an arbitrary ordering of L and represent every cluster C of L as a bit vector of length n (for every $i \in \{1, 2, \ldots, n\}$, the ith bit is set to 1 if and only if the ith leaf label belongs to C). Then, spend $O(kn^2)$ time to construct a list of bit vectors for all $O(kn)$ clusters occurring in $\mathcal{S}$ by a bottom-up traversal of each tree in $\mathcal{S}$, sort the resulting list of bit vectors by radix sort, and traverse the sorted list to identify the number of occurrences of each cluster. All this takes $O(kn^2)$ time. An alternative method, which uses $O(k^2 n)$ time, is to initialize the weight of every node in $\mathcal{S}$ to 1 and then, for $j \in \{1, 2, \ldots, k\}$, apply Day's algorithm (see Lemma 1) with $T_{ref} = T_j$ and T ranging over all T_i with $1 \le i \le k$, $i \ne j$ to find all clusters in T that also occur in T_j and increase the weights of their nodes in T by 1. Therefore, Step 1 takes $\min\{O(kn^2), O(k^2 n)\}$ time.

Next, Steps 3 and 4 make $O(k)$ calls to the procedures `Merge_Trees` and `Filter_Clusters`. The running time of `Merge_Trees` is $O(n)$ by Lemma 3 and the running time of `Filter_Clusters` is $f(n) = \Omega(n)$, so Steps 3 and 4 take $O(k \cdot f(n))$ time. □

Lemma 8 in the next section shows that $f(n) = O(n \log^2 n)$ is possible, which yields:

Corollary 1. *Algorithm `Frequency_Difference` constructs the frequency difference consensus tree of $\mathcal{S}$ in $\min\{O(kn^2), O(kn(k + \log^2 n))\}$ time.*

4.2 Procedure `Filter_Clusters`

Recall that for any node u in any input tree T_j, its weight $w(u)$ is $|K_{\Lambda(T_j[u])}(\mathcal{S})|$. Also, $w(C) = w(u)$, where $C = \Lambda(T_j[u])$. We assume that all $w(u)$-values have been computed in a preprocessing step and are available.

Let T be a tree. For every nonempty $X \subseteq V(T)$, $lca^T(X)$ denotes the lowest common ancestor of X in T. To obtain a fast solution for `Filter_Clusters`, we need the next lemma.

Lemma 6. *Let T be a tree, let X be any cluster of $\Lambda(T)$, and let $r_X = lca^T(X)$. For any $v \in V(T)$, it holds that $X \not\smile \Lambda(T[v])$ if and only if: (1) v lies on a path from a child of r_X to some leaf belonging to X; and (2) $\Lambda(T[v]) \not\subseteq X$.*

Proof. Given T, X, r_X, and v as in the lemma statement, there are four possible cases: (i) v is a proper ancestor of r_X or equal to r_X; (ii) v lies on a path from a child of r_X to some leaf in X and all leaf descendants of v belong to X; (iii) v lies on a path from a child of r_X to some leaf in X and not all leaf descendants of v belong to X; or (iv) v is a proper descendant of r_X that does not lie on any path from a leaf in X to r_X. In case (i), $X \subseteq \Lambda(T[v])$. In case (ii), $\Lambda(T[v]) \subsetneq X$. In case (iii), $X \cap \Lambda(T[v]) \neq \emptyset$ while $X \not\subseteq \Lambda(T[v])$ and $\Lambda(T[v]) \not\subseteq X$. In case (iv), $X \cap \Lambda(T[v]) = \emptyset$. By the definition of compatible clusters, $X \not\smile \Lambda(T[v])$ if and only if case (iii) occurs. □

Lemma 6 leads to an $O(n^2)$-time method for `Filter_Clusters`, which we now briefly describe. For each node $u \in V(T_A)$ in top-down order, do the following: Let $X = \Lambda(T_A[u])$ and find all $v \in V(T_B)$ such that $X \not\smile \Lambda(T_B[v])$ in $O(n)$ time by doing bottom-up traversals of T_B to first mark all ancestors of leaves belonging to X that are proper descendants of the lowest common ancestor of X in T_B, and then unmarking all marked nodes that have no leaf descendants outside of X. By Lemma 6, $X \not\smile \Lambda(T_B[v])$ if and only if v is one of the resulting marked nodes. If $w(u) \leq w(v)$ for any such v then do a *delete* operation on u in T_A. Clearly, the total running time is $O(n^2)$. (This simple method gives $f(n) = O(n^2)$ in Theorem 2 in Section 4.1, and hence a total running time of $O(kn^2)$ for Algorithm `Frequency_Difference`.) In the rest of this section, we refine this idea to get an even faster solution for `Filter_Clusters`, summarized in Fig. 5.

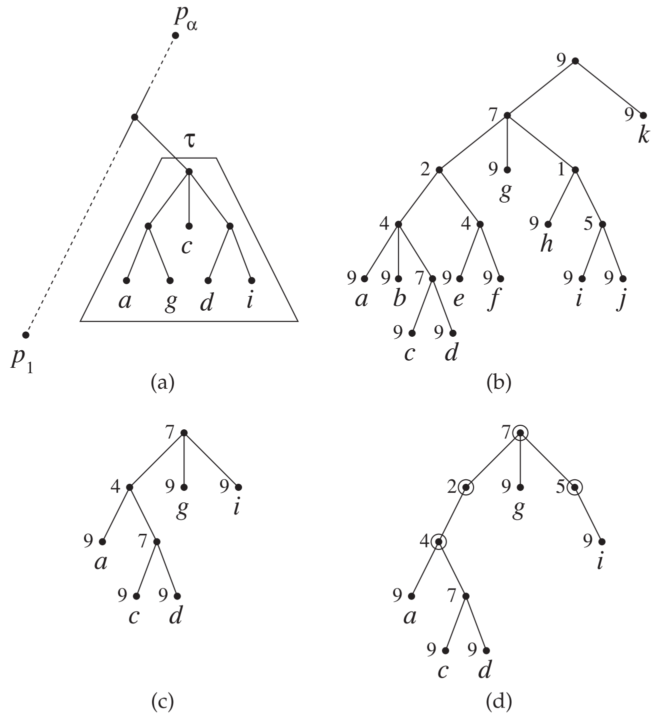
High-Level Description. We use the *centroid path decomposition* technique [30] to divide the nodes of T_A into a so-called centroid path and a set of side trees. A *centroid path* of T_A is defined as a path in T_A of the form $\pi = \langle p_\alpha, p_{\alpha-1}, \ldots, p_1 \rangle$, where p_α is the root of T_A, the node p_{i-1} for every $i \in \{2, \ldots, \alpha\}$ is any child of p_i with the maximum number of leaf descendants, and p_1 is a leaf. Given a centroid path π, removing π and all its incident edges from T_A produces a set $\sigma(\pi)$ of disjoint trees whose root nodes are children of nodes belonging to π in T_A; these trees are called the *side trees* of π. Importantly, $|\Lambda(\tau)| \leq n/2$ for every side tree τ of π. Also, $\{\Lambda(\tau) : \tau \in \sigma(\pi)\}$ forms a partition of $L \setminus \{p_1\}$. Furthermore, if π is a centroid path of T_A then the cluster collection $\mathcal{C}(T_A)$ can be written recursively as $\mathcal{C}(T_A) = \bigcup_{\tau \in \sigma(\pi)} \mathcal{C}(\tau) \cup \bigcup_{p_i \in \pi} \{\Lambda(T_A[p_i])\}$. Intuitively, this allows the cluster collection of T_A to be broken into smaller sets that can be checked more easily, and then put together again at the end.

The fast version of `Filter_Clusters` (shown in Fig. 5) first computes a centroid path $\pi = \langle p_\alpha, p_{\alpha-1}, \ldots, p_1 \rangle$ of T_A and

```
Algorithm  Filter_Clusters
Input:  Two trees $T_A$, $T_B$ with $\Lambda(T_A) = \Lambda(T_B) = L$
        such that every cluster occurring in $T_A$ or $T_B$ also
        occurs in at least one tree in $\mathcal{S}$.
Output: A tree $T$ with $\Lambda(T) = L$ and $\mathcal{C}(T) = \{\Lambda(T_A[u]) :$
        $u \in V(T_A)$ and $w(u) > w(x)$ for every $x \in$
        $V(T_B)$ with $\Lambda(T_A[u]) \not\smile \Lambda(T_B[x])\}$.
1   Compute a centroid path $\pi = \langle p_\alpha, p_{\alpha-1}, \ldots, p_1 \rangle$ of $T_A$,
    where $p_\alpha$ is the root of $T_A$ and $p_1$ is a leaf, and compute
    the set $\sigma(\pi)$ of side trees of $\pi$.

    /* Handle the side trees. */
2   Let $R_s$ be a tree consisting only of a root node and a
    single leaf labeled by $p_1$.
3   for each side tree $\tau \in \sigma(\pi)$ do
        Construct $T_B || \Lambda(\tau)$.
        Let $\tau'$ := Filter_Clusters($\tau$, $T_B || \Lambda(\tau)$).
        Attach the root of $\tau'$ as a child of the root of $R_s$.
    endfor

    /* Handle the centroid path. */
4   Let $R_c$ be a tree with $\Lambda(R_c) = L$ where every leaf is
    directly attached to the root. Do a bottom-up traversal
    of $T_B$ to precompute $|\Lambda(T_B[x])|$ for every $x \in V(T_B)$.
    Preprocess $T_B$ for lca-queries. Let $BT$ be an initially
    empty binary search tree for storing nodes from $T_B$. For
    every $x \in V(T_B)$, initialize $counter(x) := 0$.
5   Let $r_1$ := the leaf in $T_B$ labeled by $p_1$.
    Set $counter(r_1) := 1$, and if $\alpha \geq 2$ then
    $counter(parent^{T_B}(r_1)) := 1$.
6   for $i := 2$ to $\alpha$ do
6.1     Let $D := \Lambda(T_A[p_i]) \setminus \Lambda(T_A[p_{i-1}])$.
6.2     Compute $r_i := lca^{T_B}(\{r_{i-1}\} \cup D)$.
6.3     Insert every node belonging to the path between
        $r_{i-1}$ and $r_i$, except $r_{i-1}$, into $BT$.
6.4     if $counter(r_{i-1}) < |\Lambda(T_B[r_{i-1}])|$ or $r_{i-1}$ is spoiled
        then also insert $r_{i-1}$ into $BT$.
6.5     for each $x \in D$ do
            while $x$ is not in $BT$ do
                Insert $x$ into $BT$.
                $x := parent^{T_B}(x)$
            endwhile
        endfor
6.6     for each $x \in D$ do
            $counter(x) := counter(x) + 1$
            while ($counter(x) = |\Lambda(T_B[x])|$) and $x$ is not
            spoiled and $x \neq root(T_B)$ do
                $counter(parent^{T_B}(x)) :=$
                    $counter(parent^{T_B}(x)) + |\Lambda(T_B[x])|$
                Remove $x$ from $BT$.
                $x := parent^{T_B}(x)$
            endwhile
        endfor
6.7     if $r_i \in BT$ then remove $r_i$ from $BT$.
6.8     if $BT$ is empty then $M := 0$
        else $M$ := maximum weight of a node in $BT$
6.9     if $(w(\Lambda(T_A[p_i])) > M)$ then put $\Lambda(T_A[p_i])$ in $R_c$ by
        an insert operation.
    endfor

    /* Combine the surviving clusters. */
7   $T$ := Merge_Trees($R_s$, $R_c$)
8   return $T$
End Filter_Clusters
```

Fig. 5. The fast version of procedure `Filter_Clusters`.

the set $\sigma(\pi)$ of side trees of π in Step 1. Then, in Steps 2 and 3, it applies itself recursively to each side tree of π to get rid of any cluster in $\bigcup_{\tau \in \sigma(\pi)} \mathcal{C}(\tau)$ that is incompatible with some



p_α
τ
a g c d i
p_1
(a)
9 7 9 k 2 9 g 1 4 4 9 h 5 9 9 7 9 9 9 9 a b e f i j 9 9 c d
(b)
7 4 9 9 g i 9 a 7 9 9 c d
(c)
7 2 9 5 g 4 9 i 9 a 7 9 9 c d
(d)

Fig. 6. Illustrating the definition of $T_B||\Lambda(\tau)$. (a) A side tree τ of a centroid path of T_A with $\Lambda(\tau) = \{a, c, d, g, i\}$. (b) A tree T_B with a node weight displayed next to each node. (c) The tree $T_B|\{a, c, d, g, i\}$. (d) The tree $T_B||\{a, c, d, g, i\}$ with circles indicating the spoiled nodes.

cluster in T_B with a higher weight than itself, and the remaining clusters are inserted into a temporary tree R_s. Next, Steps 4, 5, and 6 check all clusters in $\bigcup_{p_i \in \pi}\{\Lambda(T_A[p_i])\}$ to determine which of them are not incompatible with any cluster in T_B with a higher weight, and create a temporary tree R_c whose cluster collection consists of all those clusters that pass this test. Finally, Step 7 combines the cluster collections of R_s and R_c by applying the procedure `Merge_Trees`. The details of Procedure `Filter_Clusters` are discussed next.

Steps 2-3 (Handling the Side Trees). For every nonempty $C \subseteq \Lambda(T)$, define $T|C$ ("the restriction of T to C"; see, e.g., [31]) as the tree T' with leaf label set C and internal node set $\{lca^T(\{a, b\}) : a, b \in C\}$ which preserves the ancestor relations from T, i.e., which satisfies $lca^T(C') = lca^{T'}(C')$ for all nonempty $C' \subseteq C$. Now, let $\sigma(\pi)$ be the set of side trees of the centroid path π of T_A computed in Step 1. For each $\tau \in \sigma(\pi)$, define a weighted tree $T_B||\Lambda(\tau)$ as follows. First, construct $T_B|\Lambda(\tau)$ and let the weight of each node in this tree equal its weight in T_B. Next, for each edge (u, v) in $T_B|\Lambda(\tau)$, let P be the path in T_B between u and v, excluding u and v; if P contains at least one node then create a new node z in $T_B|\Lambda(\tau)$, replace the edge (u, v) by the two edges (u, z) and (z, v), and set the weight of z to the maximum weight of all nodes belonging to P. See Fig. 6 for an example. Below, each such inserted node z in $T_B||\Lambda(\tau)$ is identified with any of the nodes in P that has the maximum weight.

During the construction of $T_B||\Lambda(\tau)$, if for any node u in $T_B||\Lambda(\tau)$ it holds that $(T_B||\Lambda(\tau))[u] \neq T_B[u]$ then u is marked as *spoiled*. The spoiled nodes in $T_B||\Lambda(\tau)$ are those nodes whose leaf descendant sets are missing one or more elements from $L \setminus \Lambda(\tau)$. By definition, any ancestor of a spoiled node is also a spoiled node. We extend the concept of "compatible" to nodes in $T_B||\Lambda(\tau)$ as follows. Suppose that $C \subseteq \Lambda(\tau)$. If u is a node in $T_B||\Lambda(\tau)$ and u is not spoiled then $C \smile u$ if and only if $C \smile \Lambda(T_B[u])$ holds in the original tree T_B. On the other hand, if u is a spoiled node in $T_B||\Lambda(\tau)$ then $C \smile u$ if and only if C and $\Lambda((T_B||\Lambda(\tau))[u])$ are disjoint or $C \subseteq \Lambda((T_B||\Lambda(\tau))[u])$. (Note that if u is spoiled and $\Lambda((T_B||\Lambda(\tau))[u]) \subsetneq C$ then $C \not\smile u$.) Then, for every cluster C in $\mathcal{C}(\tau)$, $\max\{w(X) : X \in \mathcal{C}(T_B) \text{ and } C \not\smile X\}$ can be expressed as $\max\{w(u) : u \text{ is a node in } T_B||\Lambda(\tau) \text{ and } C \not\smile u\}$.

In Step 3, `Filter_Clusters` is applied to $(\tau, T_B||\Lambda(\tau))$ recursively to remove all bad clusters from τ, using the spoiled nodes to pass on information regarding where in $T_B||\Lambda(\tau)$ that at least one leaf descendant not belonging to $\Lambda(\tau)$ has been pruned. For each $\tau \in \sigma(\pi)$, the resulting tree is denoted by τ'. All the clusters of each obtained τ' are inserted into a tree R_s (initially consisting of a root node and a single leaf labeled by p_1) by directly attaching the root of τ' as a child of the root of R_s. Since $\{\Lambda(\tau') : \tau \in \sigma(\pi)\}$ forms a partition of $L \setminus \{p_1\}$, every leaf label in L appears exactly once in R_s and we have $\mathcal{C}(R_s) = \{\Lambda(T_A[u]) : u \in V(\tau) \text{ for some } \tau \in \sigma(\pi) \text{ and } w(u) > w(x) \text{ for every } x \in V(T_B) \text{ with } \Lambda(T_A[u]) \not\smile \Lambda(T_B[x])\} \cup \{L\}$ after Step 3 is finished.

Steps 4, 5, and 6 (Handling the Centroid Path). The clusters $\bigcup_{p_i \in \pi}\{\Lambda(T_A[p_i])\}$ on the centroid path are nested because p_i is the parent of p_{i-1}, so $\Lambda(T_A[p_{i-1}]) \subseteq \Lambda(T_A[p_i])$ for every $i \in \{2, 3, \ldots, \alpha\}$. The main loop (Step 6) checks each of these clusters in order of increasing cardinality. For this purpose, the algorithm maintains a binary search tree *BT* that, right after Step 6.6 in any iteration i of the main loop is complete, contains nodes x from T_B with $\Lambda(T_A[p_i]) \not\smile \Lambda(T_B[x])$. Whenever a node x is inserted into *BT*, its key is set to the weight $w(T_B[x])$. Using *BT*, Step 6.8 retrieves the weight M of the heaviest cluster in T_B that is incompatible with $\Lambda(T_A[p_i])$ (if any). Step 6.9 saves $\Lambda(T_A[p_i])$ by inserting it into the tree R_c if its weight is strictly greater than M. Thus, after Step 6 is done, $\mathcal{C}(R_c) = \{\Lambda(T_A[u]) : u \in \pi \text{ and } w(u) > w(x) \text{ for every } x \in V(T_B) \text{ with } \Lambda(T_A[u]) \not\smile \Lambda(T_B[x])\}$.

In order to update *BT* correctly while moving upwards along π in Step 6, the algorithm relies on Lemma 6. In each iteration $i \in \{2, 3, \ldots, \alpha\}$ of Step 6, r_i is the lowest common ancestor in T_B of $\Lambda(T_A[p_i])$. By Lemma 6, the clusters in T_B that are incompatible with $\Lambda(T_A[p_i])$ are of the form $\Lambda(T_B[v])$, where: (1) v lies on a path in T_B from a child of r_i to a leaf in $\Lambda(T_A[p_i])$; and (2) $\Lambda(T_B[v]) \not\subseteq \Lambda(T_A[p_i])$. Accordingly, *BT* is updated in Steps 6.3, 6.4, 6.5, 6.6, and 6.7 as follows. Condition (1) is taken care of by first inserting all nodes from T_B between r_{i-1} and r_i except r_{i-1} into *BT* in Step 6.3, then also inserting r_{i-1} if $\Lambda(T_B[r_{i-1}]) \neq \Lambda(T_A[p_{i-1}])$, and finally inserting all leaf descendants of p_i that are not descendants of p_{i-1}, along with any of their ancestors in T_B that were not already in *BT*, into *BT* in Step 6.5. Lastly, Step 6.6 enforces condition (2) by using counters to locate and remove all nodes from *BT* (if any) whose clusters are proper subsets of $\Lambda(T_A[p_i])$. To do this, $counter(x)$ for every node x in T_B is updated so that it stores the number of leaves in $\Lambda(T_B[x]) \cap \Lambda(T_A[p_i])$ for the current i, and if $counter(x)$ reaches the value $|\Lambda(T_B[x])|$ then x is removed from *BT*. To account for leaf labels in the original T_B that are no longer present in the current T_B because of a recursive call in Steps 2 and 3, the algorithm exits the **while**-loop in Step 6.6 whenever it reaches a spoiled node. Hence, all encountered spoiled nodes will remain in BT after Step 6.6.

Time Complexity. To analyze the time complexity of `Filter_Clusters`, we first prove a lemma.

Lemma 7. *Let T be a tree with n leaves in which every node v has a weight $w(v)$. After $O(n \log n)$ time preprocessing, the maximum weight of all nodes on the path from any specified node in T to any specified descendant node can be recalled in $O(1)$ time.*

Proof. Decompose T into a centroid path and a set of side trees as above. Then, recursively decompose each side tree in the same way until $V(T)$ has been partitioned into a set of disjoint centroid paths. This takes $O(n)$ time according to Section 2 in [30]. Next, build two sets of data structures. First, for every centroid path $P_c = (v_1, v_2, \ldots, v_t)$, let $P_c[1..t]$ be an array of integers with $P_c[i] = w(v_i)$ for every $i \in \{1, 2, \ldots, t\}$. Store P_c in the RMQ (range minimum/maximum query) data structure of [32] which, after linear-time preprocessing, can return the index of a maximum element in the subarray $P_c[i..j]$ for any $1 \leq i \leq j \leq t$ in $O(1)$ time. Second, for every $x \in \Lambda(T)$, denote the list of all centroid subpaths contained in the path from the root of T to leaf x by $Q_1, Q_2, \ldots, Q_f$, where each Q_i is a subpath of some centroid path of T. Let $W_x[1..f]$ be an array such that $W_x[i] = \max_{v \in Q_i} w(v)$ for every $i \in \{1, 2, \ldots, f\}$. Each $W_x[i]$-entry is obtained from the P_c-RMQ data structures in $O(1)$ time, so we construct another RMQ data structure to store W_x in $O(f)$ time, and since $f = O(\log n)$, this takes $O(n \log n)$ time in total for all $x \in \Lambda(T)$.

Then, to find the maximum weight along the path from any node u to a descendant v, let $Q_1, Q_2, \ldots, Q_f$ be the concatenation of subpaths of centroid paths of T that lead from u to v. The values $\max_{v \in Q_1} w(v)$ and $\max_{v \in Q_f} w(v)$ are found in $O(1)$ time by querying the P_c-RMQ data structures for the centroid paths that contain Q_1 and Q_f, respectively, and $\max_{v \in Q_2 \cup \ldots \cup Q_{f-1}} w(v)$ is found in $O(1)$ time by querying the RMQ data structure for W_x for any $x \in \Lambda(T[v])$. □

Lemma 8. *Procedure `Filter_Clusters` runs in $O(n \log^2 n)$ time.*

Proof. Step 1 is straightforward and takes $O(n)$ time [30].

As for Steps 2 and 3, using Lemma 5.2 in [31] to first construct $T_B|\Lambda(\tau)$ for every side tree τ of π takes $O(n)$ time in total. After that, $T_B||\Lambda(\tau)$ for each side tree τ of π is obtained from $T_B|\Lambda(\tau)$ in $O(|\Lambda(\tau)| \cdot \log |\Lambda(\tau)|)$ time by applying Lemma 7, so this construction takes a total of $O(n \log n)$ time. In addition, Step 3 makes a recursive call for each τ.

Steps 4, 5, and 6 take $O(n \log n)$ time because every operation involving BT takes $O(\log n)$ time and because T_B can be preprocessed in $O(n)$ time so that any $lca^T(\{a, b\})$-query with $a, b \in L$ can be answered in $O(1)$ time [32], [33].

Finally, Step 7 takes $O(n)$ time according to Lemma 3.

For any side tree τ of π, let $g(\tau)$ represent the running time of `Filter_Clusters`(τ, $T_B||\Lambda(\tau)$). Then the total running time can be written as $O(n \log n) + \sum_{\tau \in \sigma(\pi)} g(\tau)$. Every side tree τ satisfies $|\Lambda(\tau)| \leq n/2$, so there are $O(\log n)$ recursion levels and the total running time is $O(n \log^2 n)$. □

5 Implementations

As noted in Section 1.2, there does not seem to be any publicly available implementation for the majority rule (+) consensus tree. To fill this void, we implemented algorithm `Maj_Rule_Plus` from Section 3. The situation for the frequency difference consensus tree is less critical as there already exists an implementation in the software package TNT [25]; however, TNT is very slow for large inputs, so we implemented algorithm `Frequency_Difference` from Section 4 as an alternative. Our implementations were written in C++, using some C++ libraries from Boost [34], and follow the descriptions in this article with a few exceptions (see Section 5.1). They have been included in the source code of the FACT (Fast Algorithms for Consensus Trees) package [13] which can be downloaded from:

`http://compbio.ddns.comp.nus.edu.sg/~consensus.tree/`

To test the implementations, we measured their running times for randomly generated inputs of various sizes. The experiments and their outcomes are described below.

5.1 Setup

The experiments were carried out on a Dell Optiplex 990 desktop computer running Ubuntu 15.10 and equipped with 8 GB of RAM and an Intel i7-2600 quad-core processor clocked at 3.40 GHz. The compiler was `g++`, version 5.2.1. Running times were measured using the bash `time` command.

The following methods were evaluated:

- Algorithm `Maj_Rule_Plus` from Section 3.
- Algorithm `Frequency_Difference` from Section 4.
- The "freqdifs" method in TNT [25] for computing the frequency difference consensus tree.
- The majority rule consensus tree algorithm from [13] (implementation available in the FACT package [13]).

For Step 1 of `Frequence_Difference`, we implemented both ways of computing the weights of all the clusters mentioned in the proof of Theorem 2. From here on, they are called W1 and W2, where W1 corresponds to the $O(k^2 n)$-time method and W2 to the $O(kn^2)$-time method. We also implemented the two versions of procedure `Filter_Clusters` with time complexity $O(n^2)$ (see the paragraph after Lemma 6) and $O(n \log^2 n)$ (see Lemma 8); these implementations are henceforth referred to as FC1 and FC2, respectively. (Thus, a total of four different variants of `Frequence_Difference` were included in the experiments: W1+FC1, W1+FC2, W2+FC1, and W2+FC2.) We remark here that in W2, we replaced the radix sort by quicksort, leading to an increase in its theoretical time complexity but great improvements in the actual running times. Also, although the fast version of `Filter_Clusters` is fully deterministic according to the description in Section 4.2, the implementation of it in FC2 is not, as the latter uses hash maps in order to construct the $T_B||\Lambda(\tau)$-trees more efficiently in practice.

In TNT, before running "freqdifs", the input trees were converted to its native format and the command "collapse -" was issued to prevent TNT from collapsing edges of length 0 (by default, all edges have length 0).

TABLE 1
Scenario 1 with $k = 100$ (Fixed) and Varying Values of n

n	TNT	W1+ FC1	W1+ FC2	W2+ FC1	W2+ FC2	Maj	Maj (+)
500	0.88	0.81	1.09	0.78	1.07	0.08	0.16
1000	3.74	3.37	2.46	3.33	2.43	0.15	0.26
1500	9.12	8.39	4.10	8.37	4.09	0.21	0.36
2000	17.86	16.25	6.04	16.27	6.03	0.26	0.45
3000	44.51	42.78	10.28	42.80	10.34	0.37	0.67
4000	88.43	83.02	14.66	83.31	14.84	0.47	0.89
5000	151.04	140.54	19.70	140.94	20.01	0.58	1.13
7500	421.27	374.46	35.39	379.66	36.26	0.85	1.71
10000	894.10	770.00	53.12	777.05	54.75	1.13	2.45

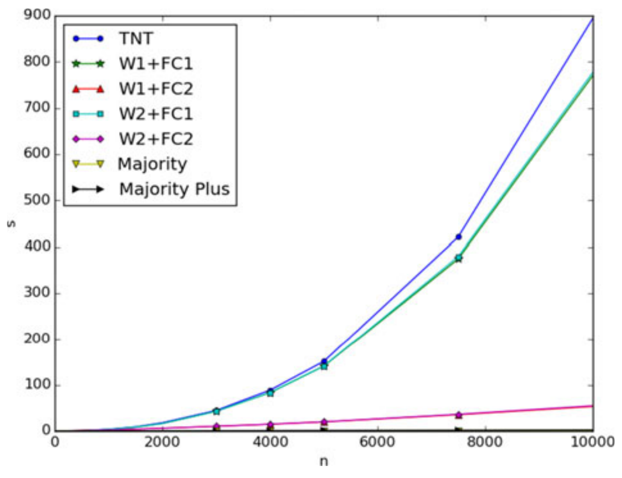


TABLE 2
Scenario 1 with $n = 100$ (Fixed) and Varying Values of k

k	TNT	W1+ FC1	W1+ FC2	W2+ FC1	W2+ FC2	Maj	Maj (+)
500	0.54	0.46	1.12	0.32	1.01	0.09	0.15
1000	1.75	1.10	2.48	0.60	2.00	0.16	0.26
1500	3.70	2.00	4.08	0.89	2.97	0.22	0.36
2000	6.38	3.17	5.89	1.18	3.90	0.27	0.46
3000	12.87	6.31	10.61	1.75	6.06	0.38	0.68
4000	20.93	10.81	16.45	2.31	8.00	0.49	0.88
5000	32.84	16.66	23.66	2.94	9.99	0.60	1.09
7500	66.40	36.03	46.47	4.43	14.98	0.87	1.60
10000	115.01	62.29	76.64	5.78	20.24	1.12	2.12

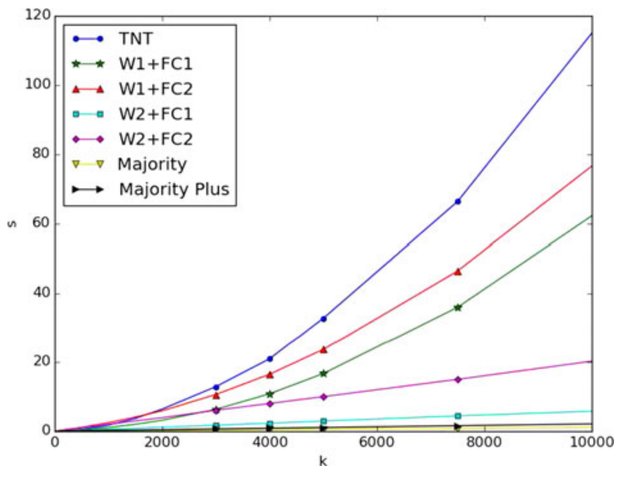


TABLE 3
Scenario 2 with $k = 100$ (Fixed) and Varying Values of n

n	TNT	W1+ FC1	W1+ FC2	W2+ FC1	W2+ FC2	Maj	Maj (+)
500	2.62	0.59	0.66	0.58	0.63	0.08	0.14
1000	10.74	2.29	1.43	2.30	1.41	0.16	0.25
1500	26.46	5.28	2.33	5.36	2.31	0.22	0.35
2000	52.35	9.55	3.33	9.72	3.35	0.26	0.45
3000	119.00	21.53	5.76	21.91	5.86	0.37	0.65
4000	227.28	37.31	8.57	37.82	8.80	0.47	0.89
5000	382.03	57.18	11.74	57.97	12.13	0.58	1.10
7500	958.78	124.05	20.74	125.37	21.65	0.85	1.70
10000	1784.01	217.55	31.84	219.53	33.50	1.16	2.37

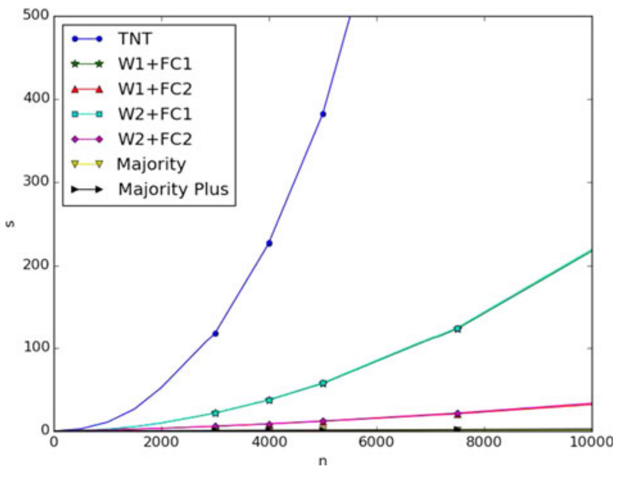
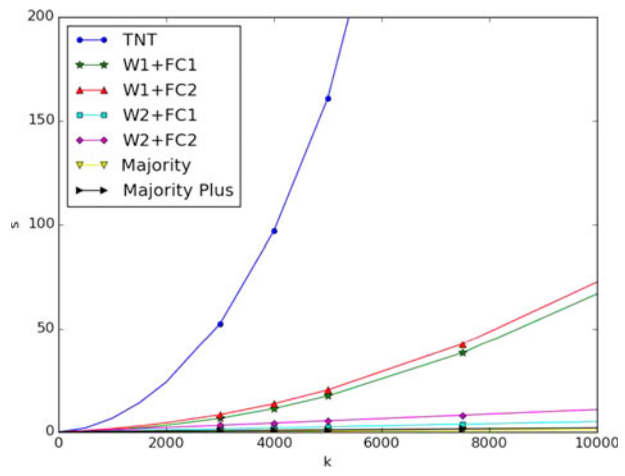


TABLE 4
Scenario 2 with $n = 100$ (Fixed) and Varying Values of k

k	TNT	W1+ FC1	W1+ FC2	W2+ FC1	W2+ FC2	Maj	Maj (+)
500	1.90	0.42	0.69	0.27	0.57	0.09	0.15
1000	6.60	1.06	1.64	0.50	1.09	0.16	0.25
1500	14.02	2.01	2.88	0.75	1.63	0.22	0.35
2000	24.03	3.23	4.40	1.00	2.16	0.27	0.44
3000	52.55	6.58	8.33	1.49	3.23	0.37	0.64
4000	97.05	11.27	13.57	1.98	4.30	0.47	0.83
5000	160.89	17.33	20.24	2.48	5.41	0.57	1.02
7500	411.02	38.18	42.50	3.76	8.09	0.82	1.51
10000	782.40	66.92	72.66	5.00	10.82	1.06	1.99



In the experiments, we measured the running times of all the above methods, averaged over 50 randomly generated inputs of size (k, n) for various specified values of (k, n). The inputs were generated according to two different scenarios, called “Scenario 1” and “Scenario 2”. Scenario 1 represents the situation where the input trees are closely related (which one may assume will occur in practice) while Scenario 2 corresponds to an extreme case in which the input trees are uncorrelated. The main difference between them is that the total number of distinct clusters can be much greater in Scenario 2. To be precise, the procedures used to generate a set of k trees over the leaf label set $\{1, 2, \dots, n\}$, for any pair of specified positive integers k and n, were:

- *Scenario 1:* First, a single binary tree with leaf label set $\{1, 2, \dots, n\}$ is generated in the uniform model [35]. Then, for each non-root, internal node u, a $delete$ operation is performed on u with probability 0.2 to obtain a non-binary tree T_r. Finally, k trees are obtained from T_r by repeating the following steps k times: take a copy of T_r and $0.05 \cdot n$ times randomly select a non-root node u and an internal node v, remove the subtree rooted at u, and attach it to v.
- *Scenario 2:* First, k binary trees with leaf label set $\{1, 2, \dots, n\}$ are generated in the uniform model [35], independently of each other. Next, for each non-root, internal node u in each tree, a $delete$ operation is performed on u with probability 0.2, yielding k non-binary trees.

5.2 Experimental Results

The obtained average running times (in seconds) are reported in Tables 1, 2, 3, and 4. Tables 1 and 2 refer to Scenario 1, and Tables 3 and 4 to Scenario 2.

According to the obtained results, the majority rule consensus tree method from [13] is faster than all the other ones. But since the majority rule consensus tree is not always informative enough (see Section 1.2), it is reassuring to see that the potentially more meaningful majority rule (+) consensus tree can be computed at a modest increase in the running time.

For computing the frequency difference consensus tree, TNT is the slowest method. The increase in its running time as the input size gets larger is particularly bad in Scenario 2, demonstrating that this method is much more vulnerable to having a huge number of distinct clusters than the others. In contrast, `Frequency_Difference` actually performs better in Scenario 2 than in Scenario 1. The reason is that trees (over a fixed leaf label set) randomly generated independently of each other tend to have many incompatible pairs of clusters, each with very few occurrences, so not many clusters will remain after `Frequency_Difference` applies `Filter_Clusters` in each iteration of its main loop. Therefore, the tree T maintained during its execution is typically much smaller in Scenario 2 than in Scenario 1, making the method run faster. We also see that the best combination of W1, W2, FC1, and FC2 to use in `Frequency_Difference` depends on (k, n). E.g., as one might expect, FC1 is faster than FC2 for small values of n due to smaller constants hidden in the big O-notation while FC2 is clearly faster than FC1 for large n. Note that Tables 1 and 3 (fixed k and varying n) suggest that the rate of growth of W2's running time is not really quadratic in n, but more like linear and very close to that of W1; nevertheless, additional (informal) experiments indicate that the memory usage of W2 grows quadratically with n, which can be explained by the algorithm using n bits for each cluster occurring in $\mathcal{S}$. Based on the experimental results, we offer the following rules of thumb: Use W2 if memory is not an issue and W1 otherwise, and use FC1 for $n < 1{,}000$ and FC2 for $n \geq 1{,}000$.

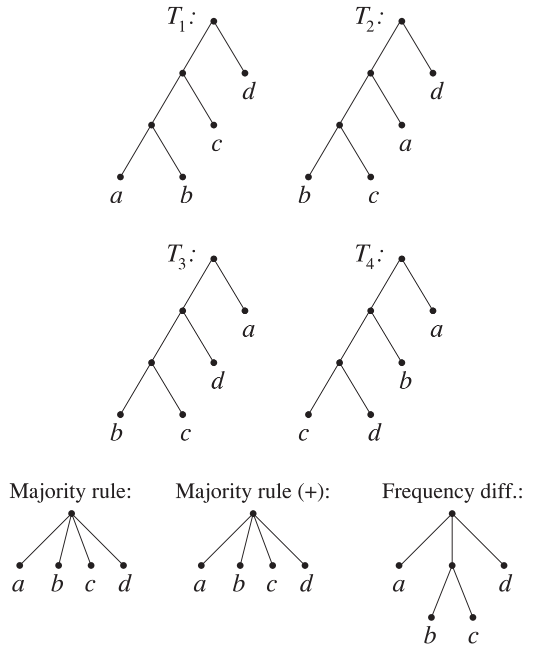


Fig. 7. Let $\mathcal{S} = \{T_1, T_2, T_3, T_4\}$ be the binary trees shown above with $L = \Lambda(T_1) = \Lambda(T_2) = \Lambda(T_3) = \Lambda(T_4) = \{a, b, c, d\}$. Then, the cluster $\{b, c\}$ is a frequency difference cluster but not a majority (+) cluster.

6 Concluding Remarks

We have presented new algorithms for constructing the majority rule (+) and frequency difference consensus trees. The time complexity of our algorithm for the majority rule (+) consensus tree is optimal because it matches the size of the input. An open problem is to further improve the running time of procedure `Filter_Clusters` in Section 4. If it is reduced to $O(n)$ then the time complexity of algorithm `Frequency_Difference` becomes $\min\{O(kn^2), O(k^2n)\}$, according to Theorem 2. This would be better than the current bound given in Corollary 1 for small k, that is, when $k = o(\log^2 n)$. An even more challenging open problem is to design an algorithm for computing the frequency difference consensus tree in optimal $O(kn)$ time.

We conclude this article with a simple observation. As shown in Fig. 1 in Section 1.1, the majority rule, majority rule (+), and frequency difference consensus trees may be different from each other. However, in the special case where all trees in $\mathcal{S}$ are *binary*, we have the following:

Lemma 9. *If $\mathcal{S} = \{T_1, T_2, \dots, T_k\}$ is a set of binary trees with $\Lambda(T_1) = \Lambda(T_2) = \dots = \Lambda(T_k) = L$ then the majority rule and the majority rule (+) consensus trees are equal, but the frequency difference consensus tree may be different.*

Proof. Consider any $C \subseteq L$ and any tree $T_i \in \mathcal{S}$. We first show that in the binary tree setting, if $C \notin \mathcal{C}(T_i)$ then $C \not\smile T_i$.

Suppose $C \notin \mathcal{C}(T_i)$. Let u be the lowest common ancestor in T_i of all leaves belonging to C. Then $C \neq \Lambda(T_i[u])$, so $\Lambda(T_i[u]) \setminus C$ is nonempty. Select any leaf $x \in \Lambda(T_i[u]) \setminus C$ and let v be the child of u that is an ancestor of x. Since T_i is binary, $T_i[v]$ must contain at least one leaf that also belongs to C (otherwise, u would not be the lowest common ancestor of C), and we have $\Lambda(T_i[v]) \cap C \neq \emptyset$, $\Lambda(T_i[v]) \not\subseteq C$, and $C \not\subseteq \Lambda(T_i[v])$. By definition, $C \not\leftarrow T_i$.

Thus, for each $T_i \in \mathcal{S}$, either $C \in \mathcal{C}(T_i)$ or $C \not\leftarrow T_i$ holds. This gives $|K_C(\mathcal{S})| + |Q_C(\mathcal{S})| = k$. Next, if C is a majority (+) cluster of $\mathcal{S}$ then $|K_C(\mathcal{S})| > |Q_C(\mathcal{S})|$ together with $|K_C(\mathcal{S})| + |Q_C(\mathcal{S})| = k$ implies $|K_C(\mathcal{S})| > \frac{k}{2}$, so C is a majority cluster of $\mathcal{S}$.

In contrast, an example of a set of binary trees whose frequency difference consensus tree differs from the majority rule and majority rule (+) consensus trees is given in Fig. 7. □

By Lemma 9, if all the input trees are binary then one can just apply the fast algorithm for the majority rule consensus tree from [13] to directly obtain the majority rule (+) consensus tree, but computing the frequency difference consensus tree is not so easy even in this special case.

ACKNOWLEDGMENTS

J.J. was funded by The Hakubi Project at Kyoto University and KAKENHI grant numbers 23700011 and 26330014. A preliminary version of this article appeared in the *Proceedings of the Thirteenth International Workshop on Algorithms in Bioinformatics* (WABI 2013), volume 8126 of *Lecture Notes in Computer Science*, pp. 141–155, Springer-Verlag Berlin Heidelberg, 2013.

REFERENCES

[1] N. Amenta, F. Clarke, and K. S. John, "A linear-time majority tree algorithm," in *Proc. 3rd Int. Workshop Algorithms Bioinf.*, Lecture Notes in Computer Science, vol. 2812, pp. 216–227, 2003.

[2] J. H. Degnan, M. DeGiorgio, D. Bryant, and N. A. Rosenberg, "Properties of consensus methods for inferring species trees from gene trees," *Systematic Biol.*, vol. 58, no. 1, pp. 35–54, 2009.

[3] J. Felsenstein, *Inferring Phylogenies*. Sunderland, MA, USA: Sinauer Associates, 2004.

[4] W.-K. Sung, *Algorithms in Bioinformatics: A Practical Introduction*. Boca Raton, FL, USA: Chapman & Hall/CRC, 2010.

[5] E. N. Adams III, "Consensus techniques and the comparison of taxonomic trees," *Systematic Zoology*, vol. 21, no. 4, pp. 390–397, 1972.

[6] D. Bryant, "A classification of consensus methods for phylogenetics," in *Bioconsensus*, M. F. Janowitz, F.-J. Lapointe, F. R. McMorris, B. Mirkin, and F. S. Roberts, Eds. Providence, RI, USA: American Mathematical Society, 2003, vol. 61, pp. 163–184.

[7] R. R. Sokal and F. J. Rohlf, "Taxonomic congruence in the Leptopodomorpha re-examined," *Systematic Zoology*, vol. 30, no. 3, pp. 309–325, 1981.

[8] W. H. E. Day, "Optimal algorithms for comparing trees with labeled leaves," *J. Classification*, vol. 2, no. 1, pp. 7–28, 1985.

[9] T. Margush and F. R. McMorris, "Consensus n-Trees," *Bulletin Math. Biol.*, vol. 43, no. 2, pp. 239–244, 1981.

[10] M. T. Holder, J. Sukumaran, and P. O. Lewis, "A justification for reporting the majority-rule consensus tree in Bayesian phylogenetics," *Systematic Biol.*, vol. 57, no. 5, pp. 814–821, 2008.

[11] Y. Cui, J. Jansson, and W.-K. Sung, "Polynomial-time algorithms for building a consensus MUL-tree," *J. Comput. Biol.*, vol. 19, no. 9, pp. 1073–1088, 2012.

[12] J. Jansson, Z. Li, and W.-K. Sung, "On finding the Adams consensus tree," in *Proc. 32nd Int. Symp. Theoretical Aspects Comput. Sci.*, vol. 30, pp. 487–499, 2015.

[13] J. Jansson, C. Shen, and W.-K. Sung, "Improved algorithms for constructing consensus trees," *J. ACM*, vol. 63, no. 3, 2016, Art. no. 28.

[14] J. Jansson and W.-K. Sung, "Constructing the R* consensus tree of two trees in subcubic time," *Algorithmica*, vol. 66, no. 2, pp. 329–345, 2013.

[15] J. Jansson, W.-K. Sung, H. Vu, and S.-M. Yiu, "Faster algorithms for computing the R* consensus tree," *Algorithmica*, accepted for publication, doi: 10.1007/s00453-016-0122-2.

[16] K. Bremer, "Combinable component consensus," *Cladistics*, vol. 6, no. 4, pp. 369–372, 1990.

[17] J. Felsenstein, "PHYLIP, version 3.6," Software package, Department of Genome Sciences, University of Washington, Seattle, USA, 2005.

[18] M. Lott, A. Spillner, K. T. Huber, A. Petri, B. Oxelman, and V. Moulton, "Inferring polyploid phylogenies from multiply-labeled gene trees," *BMC Evol. Biol.*, vol. 9, 2009, Art. no. 216.

[19] J. A. Cotton and M. Wilkinson, "Majority-rule supertrees," *Systematic Biol.*, vol. 56, no. 3, pp. 445–452, 2007.

[20] J. Dong, D. Fernández-Baca, F. R. McMorris, and R. C. Powers, "Majority-rule (+) consensus trees," *Math. Biosciences*, vol. 228, no. 1, pp. 10–15, 2010.

[21] P. A. Goloboff, J. S. Farris, M. Källersjö, B. Oxelman, M. J. Ramírez, and C. A. Szumik, "Improvements to resampling measures of group support," *Cladistics*, vol. 19, no. 4, pp. 324–332, 2003.

[22] M. Steel and J. D. Velasco, "Axiomatic opportunities and obstacles for inferring a species tree from gene trees," *Systematic Biol.*, vol. 63, no. 5, pp. 772–778, 2014.

[23] J.-P. Barthélemy and F. R. McMorris, "The median procedure for n-trees," *J. Classification*, vol. 3, no. 2, pp. 329–334, 1986.

[24] F. R. McMorris and R. C. Powers, "A characterization of majority rule for hierarchies," *J. Classification*, vol. 25, no. 2, pp. 153–158, 2008.

[25] P. A. Goloboff, J. S. Farris, and K. C. Nixon, "TNT, a free program for phylogenetic analysis," *Cladistics*, vol. 24, no. 5, pp. 774–786, 2008.

[26] R. Page, "COMPONENT, version 2.0," Software package, University of Glasgow, U.K., 1993.

[27] F. Ronquist and J. P. Huelsenbeck, "MrBayes 3: Bayesian phylogenetic inference under mixed models," *Bioinf.*, vol. 19, no. 12, pp. 1572–1574, 2003.

[28] J. Sukumaran and M. T. Holder, "DendroPy: A Python library for phylogenetic computing," *Bioinf.*, vol. 26, no. 12, pp. 1569–1571, 2010.

[29] D. L. Swofford, "PAUP*, version 4.0," Software package, Sinauer Associates, Inc., Sunderland, MA, USA, 2003.

[30] R. Cole, M. Farach-Colton, R. Hariharan, T. Przytycka, and M. Thorup, "An $O(n \log n)$ algorithm for the maximum agreement subtree problem for binary trees," *SIAM J. Comput.*, vol. 30, no. 5, pp. 1385–1404, 2000.

[31] M. Farach and M. Thorup, "Fast comparison of evolutionary trees," *Inform. Comput.*, vol. 123, no. 1, pp. 29–37, 1995.

[32] M. A. Bender and M. Farach-Colton, "The LCA problem revisited," in *Proc. 4th Latin Amer. Symp. Theoretical Informat.*, Lecture Notes in Computer Science, Springer-Verlag, vol. 1776, pp. 88–94, 2000.

[33] D. Harel and R. E. Tarjan, "Fast algorithms for finding nearest common ancestors," *SIAM J. Comput.*, vol. 13, no. 2, pp. 338–355, 1984.

[34] The Boost C++ Libraries. [Online]. Available: http://www.boost.org/

[35] A. McKenzie and M. Steel, "Distributions of cherries for two models of trees," *Math. Biosciences*, vol. 164, no. 1, pp. 81–92, 2000.

Jesper Jansson received the MSc degree in mathematics in 2002 and the PhD degree in computer science in 2003, both from Lund University, Sweden. He is currently an associate professor at Kyoto University, Japan. His research interests include graph algorithms, succinct data structures, and bioinformatics.

Ramesh Rajaby received the BSc and MSc degrees from the Department of Computer Science, the University of Milano-Bicocca, Italy. He is currently working toward the PhD degree at the Graduate School for Integrative Sciences and Engineering, the National University of Singapore.

Chuanqi Shen received the BSc and MSc degrees from the Department of Computer Science, Stanford University. Currently, he is a software engineer at Google.

Wing-Kin Sung received the BSc and PhD degrees from the Department of Computer Science, the University of Hong Kong. He is a professor in the Department of Computer Science, the National University of Singapore. He also works as a senior group leader in the Genome Institute of Singapore.