An Optimal Algorithm for Building the Majority Rule Consensus Tree

Jesper Jansson[1,⋆], Chuanqi Shen[2], and Wing-Kin Sung[3,4]

[1] Laboratory of Mathematical Bioinformatics (Akutsu Laboratory), Institute for Chemical Research, Kyoto University, Gokasho, Uji, Kyoto 611-0011, Japan
jj@kuicr.kyoto-u.ac.jp

[2] Stanford University, 450 Serra Mall, Stanford, CA 94305-2004, U.S.A.
shencq@stanford.edu

[3] School of Computing, National University of Singapore, 13 Computing Drive, Singapore 117417
ksung@comp.nus.edu.sg

[4] Genome Institute of Singapore, 60 Biopolis Street, Genome, Singapore 138672

Abstract. A deterministic algorithm for building the majority rule consensus tree of an input collection of conflicting phylogenetic trees with identical leaf labels is presented. Its worst-case running time is $O(nk)$, where n is the size of the leaf label set and k is the number of input phylogenetic trees. This is optimal since the input size is $\Omega(nk)$. Experimental results show that the algorithm is fast in practice.

1 Introduction

In the last 150 years, a vast number of phylogenetic trees [8,10,14,17,19] have been constructed and published in the literature. Existing phylogenetic trees may be based on different data sets or obtained by different methods, and do not always agree with each other; two trees can contain contradicting branching patterns even though their leaf label sets are identical. Also, when trying to infer a new, reliable phylogenetic tree from real data, heuristics for maximizing parsimony or resampling techniques such as bootstrapping may produce large collections of identically leaf-labeled phylogenetic trees having slightly different branching structures [2,3,7,8,19]. To deal with conflicts that arise between two or more such trees in a systematic manner, the concept of a *consensus tree* was invented [1,5]. Informally, a consensus tree is a phylogenetic tree which summarizes a given *collection* of phylogenetic trees. In addition to resolving conflicts, consensus trees may be employed to locate strongly supported groupings within a collection of trees [8] or as a basis for similarity measures between two given phylogenetic trees (measuring the similarity between phylogenetic trees is useful, e.g., when querying phylogenetic databases [3] or evaluating methods for phylogenetic reconstruction [12]).

⋆ Funded by The Hakubi Project and KAKENHI grant number 23700011.

M. Deng et al. (Eds.): RECOMB 2013, LNBI 7821, pp. 88–99, 2013.
© Springer-Verlag Berlin Heidelberg 2013

There are many ways to reconcile structural differences and remove inconsistencies in a collection of trees. Consequently, several alternative definitions of a "consensus tree" have been proposed since the 1970's.[1] In this paper, we concentrate on one particular type of consensus tree called the *majority rule consensus tree* [13], which is one of the most widely used consensus tree among practitioners, and present a new algorithm for constructing it. Our algorithm is fast in theory (it achieves optimal worst-case time complexity) and in practice. Furthermore, it is conceptually simple, relatively easy to implement, and deterministic, i.e., it does not use randomization or hash tables to keep track of clusters.

1.1 Definitions and Notation

We first give some basic definitions that will be used throughout the paper. 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. For short, phylogenetic trees will be referred to as "trees" from here on.

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 subset of $\Lambda(T)$ is called a *cluster of* $\Lambda(T)$. For any $u \in V(T)$, $T[u]$ denotes the subtree of T rooted at the node u, so that $\Lambda(T[u])$ is the set of all leaf labels of leaves that are descendants of u.[2] The *cluster collection of* T is defined as $\mathcal{C}(T) = \bigcup_{u \in V(T)} \{\Lambda(T[u])\}$. See Fig. 1 for an example. If a cluster $C \subseteq \Lambda(T)$ belongs to $\mathcal{C}(T)$, we say that C *occurs in* T.

The *majority rule consensus tree* is defined next. Let $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$ be a set of trees satisfying $\Lambda(T_1) = \Lambda(T_2) = \cdots = \Lambda(T_k) = L$ for some leaf label set L. A cluster that occurs in more than $k/2$ of the trees in $\mathcal{S}$ is a *majority cluster of* $\mathcal{S}$, and the *majority rule consensus tree of* $\mathcal{S}$ [13] is the unique tree T such that $\Lambda(T) = L$ and $\mathcal{C}(T)$ consists of all majority clusters of $\mathcal{S}$. The problem studied in this paper is:

> Given an input set $\mathcal{S}$ of trees with identical leaf label sets, compute the majority rule consensus tree of $\mathcal{S}$.

In the rest of the paper, we will use the following notation to refer to any input set of trees: $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$, $L = \Lambda(T_1) = \Lambda(T_2) = \cdots = \Lambda(T_k)$, and $k = |\mathcal{S}|$ and $n = |L|$. An example with $k = 3$ and $n = 6$ is provided in Fig. 1.

Finally, two clusters $C_1, C_2 \subseteq \Lambda(T)$ are said to be *pairwise compatible* if $C_1 \subseteq C_2$, $C_2 \subseteq C_1$, or $C_1 \cap C_2 = \emptyset$. Any cluster $C \subseteq \Lambda(T)$ is said to be *compatible with* T if C and $\Lambda(T[u])$ are pairwise compatible for every node $u \in V(T)$. For example, in Fig. 1, the cluster $\{a, c\}$ is compatible with T_1, but not compatible with any of the other trees. 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 that T_1 and T_2 are *compatible*.

[1] See reference [5], Chapter 30 in [8], or Chapter 8.4 in [19] for some surveys on consensus trees.

[2] For convenience, any node is considered to be a descendant of itself. This implies that if u is a leaf then $\Lambda(T[u])$ is a singleton set.

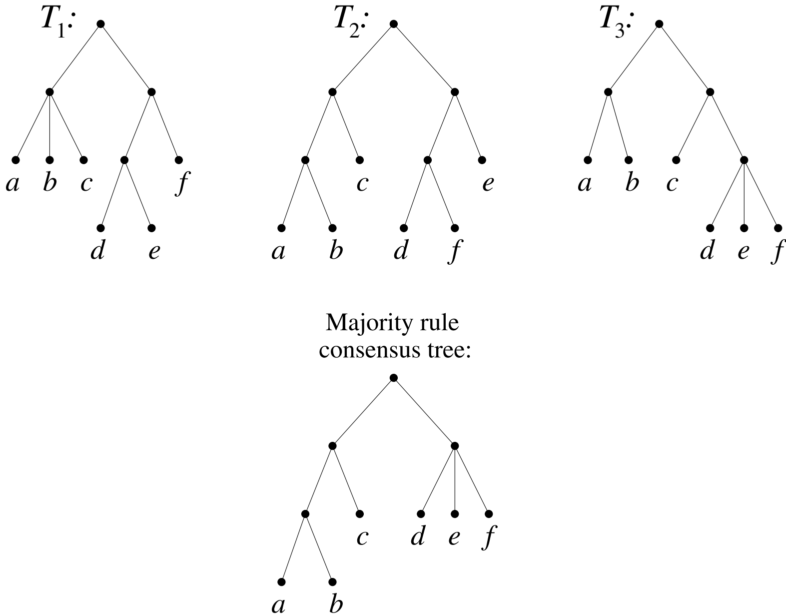

Fig. 1. In this example, $\mathcal{S} = \{T_1, T_2, T_3\}$ and $L = \Lambda(T_1) = \Lambda(T_2) = \Lambda(T_3) = \{a, b, c, d, e, f\}$. The cluster collections of T_1, T_2, and T_3 are:

$$\mathcal{C}(T_1) = \big\{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{a, b, c\}, \{d, e\}, \{d, e, f\}, L\big\},$$
$$\mathcal{C}(T_2) = \big\{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{a, b\}, \{a, b, c\}, \{d, e, f\}, \{d, f\}, L\big\},$$
$$\mathcal{C}(T_3) = \big\{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{a, b\}, \{c, d, e, f\}, \{d, e, f\}, L\big\},$$

The majority clusters of $\mathcal{S}$ are: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{a, b\}, \{a, b, c\}, \{d, e, f\}, L$.

1.2 Previous Work

The majority rule consensus tree was introduced by Margush and McMorris [13] in 1981. In 1985, Wareham [21] published a deterministic algorithm for building the majority rule consensus tree with a worst-case running time of $O(n^2 + nk^2)$. This was the record until very recently; in [11], we developed a faster deterministic algorithm with $O(nk \log k)$ worst-case running time, based on recursion.

As for *randomized* methods, Amenta *et al.* [2] gave an algorithm with expected running time $O(nk)$ but unbounded worst-case running time. Here, randomization is used to count and store the number of occurrences of clusters from $\mathcal{S}$ in suitably constructed hash tables. We note that the implementations for computing majority rule consensus trees in existing software packages such as PHYLIP [9], MrBayes [16], SumTrees in DendroPy [18], COMPONENT [15], and PAUP* [20] also rely on randomization, and typically have unbounded worst-case running times as well.

1.3 New Results and Organization of the Paper

This paper presents a deterministic algorithm for computing the majority rule consensus tree. Its worst-case running time is $O(nk)$, which is optimal because the size of the input is $\Omega(nk)$. We thus resolve a long-standing open problem in Phylogenetics.

To ensure that our algorithm is practical, we implemented it and performed a series of experiments to compare its actual running time to that of the majority rule consensus tree method in PHYLIP [9]. (We chose PHYLIP's majority rule consensus tree method as a benchmark because it is freely available, frequently used in practice, and faster than many other methods such as SumTrees in DendroPy [18] and COMPONENT [15].) The experiments showed that our deterministic method is much faster than PHYLIP for certain types of large inputs, e.g., when $n \gg k$, implying that randomization may not be necessary in many cases.

The rest of the paper is organized as follows. Section 2 summarizes a few results from the literature that are needed later. To help us find an efficient solution to the majority rule consensus tree problem, Section 3 outlines a technique for identifying all majority elements in a list $\mathcal{W}$ of subsets of a fixed set, where a *majority element* is defined to be any element that occurs in more than half of the subsets in $\mathcal{W}$. This technique is subsequently employed in our new majority rule consensus tree algorithm, named `Fast_Maj_Rule_Cons_Tree`, which is described and analyzed in Section 4. Next, Section 5 reports the running times of our prototype implementation of `Fast_Maj_Rule_Cons_Tree` when applied to some simulated data sets. Finally, the availability of the prototype implementation is discussed in Section 6.

2 Preliminaries

We shall make use of the following results from the literature. (For further details, see the respective original references.)

2.1 Day's Algorithm [6]

Day's algorithm [6] takes two trees T_{ref} and T with identical leaf label sets as input. After linear-time preprocessing, the algorithm can check whether or not any specified cluster that occurs in T also occurs in T_{ref}, and each such check can be performed in constant time.

Theorem 1. *(Day [6]) 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.*

2.2 Procedure `One-Way_Compatible` [11]

`One-Way_Compatible` is a linear-time procedure defined in Section 4.1 of [11]. Its input is two trees T_1 and T_2 with identical leaf label sets, and its output

is a copy of T_1 in which every cluster that is not compatible with T_2 has been removed. The procedure is asymmetric; for example, if T_1 consists of n leaves attached to a root node and $T_2 \neq T_1$ then `One-Way_Compatible`$(T_1, T_2) = T_1$, while `One-Way_Compatible`$(T_2, T_1) = T_2$.

Theorem 2. *([11]) Let T_1 and T_2 be two given trees with $\Lambda(T_1) = \Lambda(T_2) = L$ and let $n = |L|$. Procedure* `One-Way_Compatible`(T_1, T_2) *returns a tree T with $\Lambda(T) = L$ such that $\mathcal{C}(T) = \{C \in \mathcal{C}(T_1) : C \text{ is compatible with } T_2\}$ in $O(n)$ time.*

2.3 Procedure `Merge_Trees` [11]

The procedure `Merge_Trees` from Section 2.4 in [11] combines all the clusters from two compatible trees into one tree in linear time.

Theorem 3. *([11]) Let T_1 and T_2 be two given trees with $\Lambda(T_1) = \Lambda(T_2) = L$ that are compatible and let $n = |L|$. Procedure* `Merge_Trees`(T_1, T_2) *returns a tree T with $\Lambda(T) = L$ and $\mathcal{C}(T) = \mathcal{C}(T_1) \cup \mathcal{C}(T_2)$ in $O(n)$ time.*

2.4 The *delete* and *insert* Operations on a Tree

Let T be a tree and let u be any non-root, internal node in T. Applying the *delete* operation on u modifies T as follows: First, all children of u become children of the parent of u, and then u and the edge between u and its parent are removed. See Fig. 2 for an illustration. Note that by applying the *delete* operation on node u, the cluster $\Lambda(T[u])$ is removed from the cluster collection $\mathcal{C}(T)$ while all other clusters are preserved. Also note that the time needed for this operation is proportional to the number of children of u.

The *insert* operation is the inverse of the *delete* operation. It inserts a new internal node into T, thereby creating an additional cluster in $\mathcal{C}(T)$.

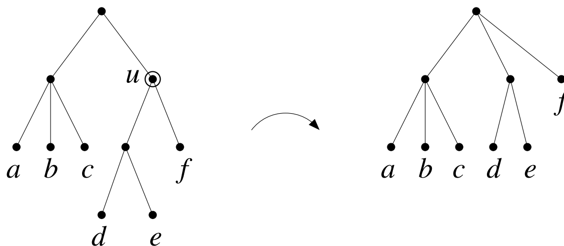

Fig. 2. Figure from [11]. Let T be the tree on the left and let u be the marked node. Then $\Lambda(T[u]) = \{d, e, f\}$ and applying the *delete* operation on u removes the cluster $\{d, e, f\}$ from $\mathcal{C}(T)$.

3 Finding All Majority Elements

In this section, we describe a technique for solving a problem closely related to the majority rule consensus tree problem: Given a list $\mathcal{W}$ of subsets of a set X, output all majority elements in $\mathcal{W}$, where a *majority element in* $\mathcal{W}$ is defined to be any element of X that occurs in more than half of the subsets in $\mathcal{W}$. It can be solved easily by using one counter for each element in X, but when $|X|$ is very large and many elements from X never occur in $\mathcal{W}$ at all, we need a method whose time complexity does not depend on $|X|$.

Denote $k = |\mathcal{W}|$, and for any $j \in \{1, 2, \ldots, k\}$, let $\mathcal{W}[j]$ be the jth subset in the list $\mathcal{W}$. For our purposes, it is sufficient to focus on the restriction of the problem in which X is an ordered set and each $\mathcal{W}[j]$ is specified as a sorted list. The following two-phase algorithm solves the restricted problem by maintaining a set of *current candidates*, which are certain elements belonging to X, along with a counter for each current candidate:

- Phase 1: Initialize the set of current candidates as the empty set. Sweep through $\mathcal{W}$, i.e., for each $j \in \{1, 2, \ldots, k\}$, consider $\mathcal{W}[j]$ and do the following. Firstly, for every current candidate x, increase x's counter by 1 if $x \in \mathcal{W}[j]$, or decrease it by 1 if $x \notin \mathcal{W}[j]$; if x's counter reaches 0 then remove x from the set of current candidates. Secondly, insert every $x \in \mathcal{W}[j]$ which is not a current candidate into the set of current candidates and initialize its counter to 1.
- Phase 2: Let X' be the set of current candidates. Sweep through $\mathcal{W}$ one more time to count the total number of occurrences in $\mathcal{W}$ of every element in X'. Output the ones that occur more than $\frac{k}{2}$ times.

As an example, let $X = \{a, b, c, d, e\}$ and $\mathcal{W} = (\mathcal{W}[1], \mathcal{W}[2], \mathcal{W}[3]) = (\{a, b, d\}, \{a, c\}, \{d, e\})$. Then the set of current candidates at the end of Phase 1 will be $\{a, d, e\}$. In Phase 2, the algorithm outputs a and d.

To prove the correctness of this method, observe that for any $x \in X$, if x occurs in more than $\frac{k}{2}$ subsets in $\mathcal{W}$, then x must be one of the current candidates at the end of Phase 1 because its counter is > 0. Hence, all majority elements in $\mathcal{W}$ (if any) belong to the set X'. However, as in the example above, some non-majority elements might also be included in X'. For this reason, Phase 2 is used to identify those elements that indeed occur more than $\frac{k}{2}$ times. To analyze the time complexity, since each $\mathcal{W}[j]$ is given as a sorted list, it is easy to maintain the set of current candidates in a sorted list and implement all operations for that value of j in time proportional to the number of current candidates. This yields:

Lemma 1. *Let X be an ordered set and let $\mathcal{W}$ be a list of sorted subsets of X. The above algorithm outputs all majority elements in $\mathcal{W}$ in $O(k \cdot y)$ time, where $k = |\mathcal{W}|$ and at most y elements from X belong to the set of current candidates at any point in time.*

Remark: Boyer and Moore's classical algorithm in [4] solves the special case of the problem where every subset in the list $\mathcal{W}$ has cardinality 1. The algorithm presented above can be viewed as an extension of [4].

4 An Optimal Algorithm for the Majority Rule Consensus Tree

This section presents the new algorithm `Fast_Maj_Rule_Cons_Tree` for building the majority rule consensus tree of an input collection $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$ of identically leaf-labeled trees, where $\Lambda(T_1) = \Lambda(T_2) = \cdots = \Lambda(T_k) = L$. It uses the technique from Section 3 to locate all majority clusters in $\mathcal{S}$ by interpreting X as the set of all possible clusters of L (so that every element $x \in X$ is a subset of L) and the list $\mathcal{W}$ as the length-k sequence of cluster collections of the trees in $\mathcal{S}$. In other words, $\mathcal{W} = (\mathcal{W}[1], \mathcal{W}[2], \ldots, \mathcal{W}[k]) = (\mathcal{C}(T_1), \mathcal{C}(T_2), \ldots, \mathcal{C}(T_k))$, and for every $j \in \{1, 2, \ldots, k\}$, it holds that $\mathcal{W}[j] \subseteq X$.

Algorithm `Fast_Maj_Rule_Cons_Tree` also consists of two phases. In Phase 1, it finds all clusters that might be majority clusters, and then, in Phase 2,

```
Algorithm  Fast_Maj_Rule_Cons_Tree
Input:   A collection S = {T_1, T_2, ..., T_k} of trees with Λ(T_1) = Λ(T_2) = ··· =
         Λ(T_k).
Output:  The majority rule consensus tree of S.

    /* Phase 1 */
1   T := T_1
2   for each v ∈ V(T) do count(v) := 1
3   for j := 2 to k do
3.1     for each v ∈ V(T) in top-down order do
            if Λ(T[v]) occurs in T_j then count(v) := count(v) + 1
            else count(v) := count(v)−1; if count(v) reaches 0 then delete node v.
        endfor
3.2     for every cluster C in T_j that is compatible with T but does not occur
        in T do
            Insert C into T.
            Initialize count(v) := 1 for the new node v satisfying Λ(T[v]) = C.
        endfor
    endfor

    /* Phase 2 */
4   for each v ∈ V(T) do count(v) := 0
5   for j := 1 to k do
5.1     for each v ∈ V(T) do
            if Λ(T[v]) occurs in T_j then count(v) := count(v) + 1
6   for each v ∈ V(T) in top-down order do
        if count(v) ≤ k/2 then perform a delete operation on v.
7   return T
End  Fast_Maj_Rule_Cons_Tree
```

Fig. 3. The pseudocode for Algorithm `Fast_Maj_Rule_Cons_Tree`

eliminates those candidates that do not occur in more than $\frac{k}{2}$ of the trees in $\mathcal{S}$. Whatever clusters that remain must be the majority clusters of $\mathcal{S}$. During the algorithm's execution, the current candidates are stored as nodes in a tree T, as explained below.

The pseudocode is summarized in Fig. 3. Phase 1 and Phase 2 are described in Sections 4.1 and 4.2, respectively. To achieve a good time complexity, some steps of the algorithm are implemented by applying Day's algorithm [6] and the procedures `One-Way_Compatible` and `Merge_Trees` mentioned in Section 2; the details are given in Section 4.3.

4.1 Description of Phase 1

Phase 1 of the algorithm examines the trees $T_1, T_2, \ldots, T_k$ in sequential order. As in Section 3, the algorithm maintains a set of current candidates, each equipped with its own counter. Every current candidate is some cluster of L and thus an element from X, like before. However, there are two crucial differences between `Fast_Maj_Rule_Cons_Tree` and the method in Section 3.

The first difference is that `Fast_Maj_Rule_Cons_Tree` does not store the set of current candidates in a sorted list as in Section 3, but encodes them as *nodes in a tree* T whose leaf label set equals L. (This is the key to getting an efficient algorithm.) To be precise, every node v in T represents a current candidate cluster $\Lambda(T[v])$ and has a counter $count(v)$. For any $j \in \{1, 2, \ldots, k\}$, when treating tree T_j, all clusters in $\mathcal{C}(T)$ that also belong to $\mathcal{C}(T_j)$ get their counters incremented by 1, while all clusters in $\mathcal{C}(T)$ that do not belong to $\mathcal{C}(T_j)$ get their counters decremented by 1. If this leads to some counter reaching 0 then the internal node in T corresponding to that cluster is deleted. Next, all other clusters in $\mathcal{C}(T_j)$ that are not current candidates but are compatible with T are upgraded to current candidate-status by inserting them into T and initializing their corresponding nodes' counters to 1.

The other important difference between this approach and the one in Section 3 is that for any $j \in \{2, \ldots, k\}$, a cluster C that occurs in T_j but is not a current candidate does not automatically become a current candidate; C will only be inserted into T if it is pairwise compatible with all the current candidates. We therefore need an additional lemma to guarantee the correctness of Phase 1:

Lemma 2. *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 $\mathcal{S}$. During the execution of Phase 1, for any $j \in \{1, 2, \ldots, k\}$, say that C is *blocked in iteration* j if the following happens: C is not a current candidate, C occurs in tree T_j, and C is not allowed to become a current candidate because C is not compatible with the current T.

Let a denote the number of trees in $\mathcal{S}$ in which C occurs. By the definition of a majority cluster, $a > \frac{k}{2}$. Hence, there are $k - a < \frac{k}{2}$ trees in $\mathcal{S}$ in which C does not occur. We claim that each such tree T_x can cancel out the effect on C's

counter of at most one of the a occurrences of C in $\mathcal{S}$. To prove the claim, let T_x be any tree in $\mathcal{S}$ in which C does not occur and consider the two possible cases:

- If C is a current candidate when T_x is considered, then C's counter will be decremented by 1.
- If C is not a current candidate when T_x is considered, then some clusters which are not pairwise compatible with C may get their counters incremented by 1. As a result, C may be blocked in another iteration.

Next, since $a - (k - a) > \frac{k}{2} - \frac{k}{2} = 0$, the counter for C will have a non-zero value at the end of Phase 1. By the definition of the tree T in the algorithm, $C \in \mathcal{C}(T)$ holds. □

4.2 Description of Phase 2

Phase 2 of the algorithm is straightforward. It checks how many times every cluster in the tree T occurs among $T_1, T_2, \ldots, T_k$. Any clusters that do not occur more than $\frac{k}{2}$ times are removed from T. It follows immediately from Lemma 2 that the cluster collection of the remaining tree T equals the set of all majority clusters of $\mathcal{S}$. Hence, the output of the algorithm is the majority rule consensus tree.

Lemma 3. *The tree output by Algorithm* `Fast_Maj_Rule_Cons_Tree` *at the end of Phase 2 is the majority rule consensus tree of* $\mathcal{S}$.

4.3 Time Complexity Analysis

We now analyze the worst-case time complexity of `Fast_Maj_Rule_Cons_Tree`.

Theorem 4. *Algorithm* `Fast_Maj_Rule_Cons_Tree` *constructs the majority rule consensus tree of* $\mathcal{S}$ *in* $O(nk)$ *worst-case time, where* $n = |L|$ *and* $k = |\mathcal{S}|$.

Proof. We first show that in Phase 1, every iteration of the main loop in Step 3 takes $O(n)$ time. To perform Step 3.1 in $O(n)$ time, run Day's algorithm [6] with $T_{ref} = T_j$ and then check each $\Lambda(T[v])$ to see if it occurs in T_j. By Theorem 1, this requires $O(n)$ time for preprocessing, and each of the $O(n)$ nodes in $V(T)$ can be checked in $O(1)$ time. The *delete* operations take $O(n)$ time in total since the nodes are handled in top-down order (every node is moved at most once because if some node is deleted and its children moved then these children will not need to be moved again in the same iteration). Next, Step 3.2 can be implemented in $O(n)$ time by letting $P :=$ `One-Way_Compatible`(T_j, T) and $Q :=$ `Merge_Trees`(P, T), and then updating the structure of T to make T isomorphic to the obtained Q (and setting the counters of all new nodes to 1). This works because according to Theorem 2, P is a tree consisting of the clusters occurring in T_j that are compatible with the set of current candidates, and by Theorem 3, Q is the result of inserting each such cluster into T, if it did not already occur in T. There are $O(k)$ iterations in the main loop, so Phase 1 takes $O(nk)$ time.

In Phase 2, Step 5.1 is executed in $O(n)$ time, again by applying Day's algorithm [6] with $T_{ref} = T_j$ so that each $\Lambda(T[v])$ can be checked in $O(1)$ time.[3] Thus, the loop in Step 5 takes $O(nk)$ time. Step 6 can be carried out in $O(n)$ time by treating the nodes in top-down order as above. In total, Phase 2 also takes $O(nk)$ time. □

5 Experimental Results

We implemented `Fast_Maj_Rule_Cons_Tree` in C++ and compared its worst-case and average running times to those of PHYLIP [9] and the previously fastest ($O(nk \log k)$ time) deterministic algorithm from [11] for some simulated data sets. The experiments were run on Ubuntu Nutty Narwhal, a 64-bit operating system with 8.00 GB RAM, and a 2.20 GHz CPU. Below, we refer to the majority rule consensus tree method in PHYLIP as "M-PHYLIP", the implementation of the algorithm in [11] as "M-Fast-v1", and the implementation of the new algorithm `Fast_Maj_Rule_Cons_Tree` presented in this paper as "M-Fast-v2".

We generated 10 data sets for various specified values of the parameters n and k with the method described in Section 6.2 of [11], applied the three majority consensus tree methods to each data set, and measured the running times. First, the following values of n and k were evaluated:

- (a) $n = 500$, $k = 1000$
- (b) $n = 1000$, $k = 500$
- (c) $n = 2000$, $k = 1000$
- (d) $n = 5000$, $k = 100$

The worst-case and average running times (in seconds) are reported below.

(a) $n = 500$, $k = 1000$:

	Worst-case	Average
M-PHYLIP	1.94	1.88
M-Fast-v1	8.10	8.00
M-Fast-v2	3.72	3.69

(b) $n = 1000$, $k = 500$:

	Worst-case	Average
M-PHYLIP	3.50	3.19
M-Fast-v1	7.54	7.38
M-Fast-v2	3.80	3.67

(c) $n = 2000$, $k = 1000$:

	Worst-case	Average
M-PHYLIP	34.07	30.03
M-Fast-v1	32.24	31.96
M-Fast-v2	16.09	14.86

(d) $n = 5000$, $k = 100$:

	Worst-case	Average
M-PHYLIP	93.25	90.04
M-Fast-v1	6.41	6.27
M-Fast-v2	4.40	4.28

[3] This way of counting occurrences of clusters has been used elsewhere in the literature, e.g., in [21] and on p. 217 of [19].

The experimental results indicate that `Fast_Maj_Rule_Cons_Tree` is exceptionally useful when n is large. For example, when $n = 5000$ and $k = 100$, it is about 20 times faster than M-PHYLIP. On the other hand, M-PHYLIP is faster in practice for inputs with $n \ll k$.

Next, we tried the methods on some even bigger inputs. M-PHYLIP returned "Error allocating memory" for $n = 2000$, $k \geq 2000$, whereas M-Fast-v2 worked fine and obtained the following worst-case and average running times.

(e) $n = 2000$, $k = \{2000, 3000, 4000, 5000\}$:

k	Worst-case	Average
2000	31.22	30.86
3000	47.42	46.23
4000	62.54	61.88
5000	78.96	77.78

This shows that `Fast_Maj_Rule_Cons_Tree` may come in handy when analyzing large phylogenetic data sets.

6 Concluding Remarks

We have proved that the majority rule consensus tree can be built in (optimal) $O(nk)$ time in the worst case, without using randomization. Although this might at first appear to be a purely theoretical result, it has practical implications as well. The experiments demonstrated that our deterministic algorithm `Fast_Maj_Rule_Cons_Tree` is much faster than randomized methods such as the one found in PHYLIP [9] when the input trees are very large, i.e., when $n \gg k$. In contrast to current practice, this suggests that it might not always be a good idea to use randomization and hashing when computing majority rule consensus trees.

We hope that the new algorithm will be a helpful tool for bioinformaticians working with huge phylogenetic trees in the future. We have included it in the FACT (Fast Algorithms for Consensus Trees) package [11] at:

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

The C++ source code of our prototype implementation used in Section 5 can also be downloaded from the same webpage.

References

1. Adams III., E.N.: Consensus techniques and the comparison of taxonomic trees. Systematic Zoology 21(4), 390–397 (1972)
2. Amenta, N., Clarke, F., St. John, K.: A Linear-Time Majority Tree Algorithm. In: Benson, G., Page, R.D.M. (eds.) WABI 2003. LNCS (LNBI), vol. 2812, pp. 216–227. Springer, Heidelberg (2003)

3. Bansal, M.S., Dong, J., Fernández-Baca, D.: Comparing and aggregating partially resolved trees. Theoretical Computer Science 412(48), 6634–6652 (2011)
4. Boyer, R.S., Moore, J.S.: MJRTY – A Fast Majority Vote Algorithm. In: Boyer, R.S. (ed.) Automated Reasoning: Essays in Honor of Woody Bledsoe. Automated Reasoning Series, pp. 105–117. Kluwer Academic Publishers (1991)
5. Bryant, D.: A classification of consensus methods for phylogenetics. In: Janowitz, M.F., Lapointe, F.-J., McMorris, F.R., Mirkin, B., Roberts, F.S. (eds.) Bioconsensus. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 61, pp. 163–184. American Mathematical Society (2003)
6. Day, W.H.E.: Optimal algorithms for comparing trees with labeled leaves. Journal of Classification 2(1), 7–28 (1985)
7. Degnan, J.H., DeGiorgio, M., Bryant, D., Rosenberg, N.A.: Properties of consensus methods for inferring species trees from gene trees. Systematic Biology 58(1), 35–54 (2009)
8. Felsenstein, J.: Inferring Phylogenies. Sinauer Associates, Inc., Sunderland (2004)
9. Felsenstein, J.: PHYLIP, version 3.6. Software package, Department of Genome Sciences, University of Washington, Seattle, U.S.A. (2005)
10. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, New York (1997)
11. Jansson, J., Shen, C., Sung, W.-K.: Improved algorithms for constructing consensus trees. In: Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, pp. 1800–1813. SIAM (2013)
12. Kuhner, M.K., Felsenstein, J.: A simulation comparison of phylogeny algorithms under equal and unequal evolutionary rates. Molecular Biology and Evolution 11(3), 459–468 (1994)
13. Margush, T., McMorris, F.R.: Consensus n-Trees. Bulletin of Mathematical Biology 43(2), 239–244 (1981)
14. Nakhleh, L., Warnow, T., Ringe, D., Evans, S.N.: A comparison of phylogenetic reconstruction methods on an Indo-European dataset. Transactions of the Philological Society 103(2), 171–192 (2005)
15. Page, R.: COMPONENT, version 2.0. Software package. University of Glasgow, U.K. (1993)
16. Ronquist, F., Huelsenbeck, J.P.: MrBayes 3: Bayesian phylogenetic inference under mixed models. Bioinformatics 19(12), 1572–1574 (2003)
17. Semple, C., Steel, M.: Phylogenetics. Oxford Lecture Series in Mathematics and its Applications, vol. 24. Oxford University Press (2003)
18. Sukumaran, J., Holder, M.T.: DendroPy: a Python library for phylogenetic computing. Bioinformatics 26(12), 1569–1571 (2010)
19. Sung, W.-K.: Algorithms in Bioinformatics: A Practical Introduction. Chapman & Hall/CRC (2010)
20. Swofford, D.L.: PAUP*, version 4.0. Software package. Sinauer Associates, Inc., Sunderland (2003)
21. Wareham, H.T.: An efficient algorithm for computing M_l consensus trees. B.Sc. Honours thesis, Memorial University of Newfoundland, Canada (1985)