Improved Algorithms for Constructing Consensus Trees

Jesper Janssont
Chuanqi Shen‡
Wing-Kin Sung‡

Abstract

A consensus tree is a single phylogenetic tree that summarizes the branching structure in a given set of conflicting phylogenetic trees. Many different types of consensus trees have been proposed in the literature; three of the most well-known and widely used ones are the majority rule consensus tree, the loose consensus tree, and the greedy consensus tree. This paper presents new deterministic algorithms for constructing them that are faster than all the previously known ones. Given $k$ phylogenetic trees with $n$ leaves each and with identical leaf label sets, our algorithms run in $O(nk \log k)$ time (majority rule consensus tree), $O(nk)$ time (loose consensus tree), and $O(n^2k)$ time (greedy consensus tree).

1 Introduction

Scientists and scholars often use phylogenetic trees to describe evolutionary relationships [9, 11, 19, 22, 25]. For various reasons, inferring an accurate phylogenetic tree can be a difficult task, and in some settings (such as when using multiple data sets, when applying resampling techniques like bootstrapping to the same data set, or when using heuristics for maximizing parsimony), one ends up with a large collection $S$ of trees rather than a single tree [1, 2, 8, 9, 14, 25]. However, it might be necessary to represent all of $S$ by one tree, even though the branching structures of the trees in $S$ are in varying degrees of conflict with each other. In this case, a consensus tree is used to summarize them. Depending on the application and the quality of the input data, different definitions of a “consensus tree” may be appropriate. As a result, during the last 40 years, many alternative types of consensus trees have been introduced and analyzed by biologists, mathematicians, and computer scientists; see, e.g., [5], Chapter 30 in [9], or Chapter 8.4 in [25] for some surveys.

Three frequently used types of consensus trees are: (i) the majority rule consensus tree [16], (ii) the loose consensus tree [4], and (iii) the greedy consensus tree [5, 10]. (The loose consensus tree is also known in the literature as the semi-strict consensus tree or the combinatorial component consensus tree, and the greedy consensus tree is also known as the majority rule extended consensus tree.) For example, a search on Google Scholar for “majority rule consensus tree” returns thousands of articles published in biology-related journals using this concept. Popular computational phylogenetics software packages such as PHYLIP [10] and MrBayes [21] contain implementations for constructing (i) and (iii), COMPONENT [20] implements (i) and (ii), SumTrees in DendroPy [24] implements (i), and PAUP* [26] implements (i), (ii), and (iii). Although these programs work very well in practice, they rely on randomization and their worst-case running times may be unbounded. On the other hand, the fastest deterministic algorithms published in the literature are quite slow. This situation is unsatisfactory from a theoretical point of view. In this paper, we develop new, simple deterministic algorithms for constructing (i), (ii), and (iii) that are faster in the worst case than the currently best published ones. In particular, our algorithm for (ii) is asymptotically optimal.

1.1 Definitions and notation. 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. To simplify the presentation, phylogenetic trees are referred to as “trees” from here on, and every leaf in a tree is identified with its (unique) label. If $u$ and $v$ are nodes in a tree and there is a directed path from $u$ to $v$ then $u$ is an ancestor of $v$, and $v$ is a descendant of $u$. Every node in a tree $T$ is considered to be an ancestor as well as a descendant of itself; for any nodes $u$, $v$ in $T$, in case $v$ is a descendant of $u$ and $u \neq v$ then we call $v$ a proper descendant of $u$. For any non-empty subset $S$ of nodes in a tree $T$, the

---

tLaboratory of Mathematical Bioinformatics (Akutsu Laboratory), Institute for Chemical Research, Kyoto University, Gokasho, Uji, Kyoto 611-0011, Japan. Funded by The Hakubi Project at Kyoto University and KAKENHI grant number 23700011. E-mail: jj@kuscr.kyoto-u.ac.jp

‡Currently enrolled at Stanford University, 450 Serra Mall, Stanford, CA 94305-2004, U.S.A. At the time this paper was written, C. Shen was a high school student at Raffles Institution, 1 Raffles Institution Lane, Singapore 579594. E-mail: scq1993@gmail.com

∗School of Computing, National University of Singapore, 13 Computing Drive, Singapore 117417. Also affiliated with Genome Institute of Singapore, 60 Biopolis Street, Genome, Singapore 138672. E-mail: ksung@comp.nus.edu.sg
lowest common ancestor of $S$ in $T$, denoted by $\text{lea}^T(S)$, is the unique node $w$ in $T$ such that: (i) $w$ is an ancestor of all nodes in $S$; and (ii) $w$ has no proper descendant that is an ancestor of all nodes in $S$.

Let $T$ be a tree. The set of all nodes in $T$ is denoted by $V(T)$ and the set of all leaves in $T$ by $\Lambda(T)$. Any subset $C$ of $\Lambda(T)$ is called a cluster of $\Lambda(T)$. For any $u \in V(T)$, the subtree of $T$ rooted at $u$ (i.e., the subgraph of $T$ induced by the set of descendants of $u$) is written as $T[u]$, and $\Lambda(T[u])$ is called the cluster associated with $u$. Thus, the cluster associated with a node $u$ consists of the descendants of $u$ that are leaves. The cluster collection of $T$ is defined as $C(T) = \bigcup_{u \in V(T)} \Lambda(T[u])$. If a cluster $C \subseteq \Lambda(T)$ belongs to $C(T)$, we say that $C$ occurs in $T$. Two clusters $C_1, C_2 \subseteq \Lambda(T)$ are called 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 Figure 1, the cluster $\{d, e\}$ occurs in $T_1$ but not in $T_2$ and $T_3$; however, $\{d, e\}$ is compatible with $T_2$ and $T_3$. For any $C \subseteq \Lambda(T)$, if $|C| = 1$ or $C = \Lambda(T)$ then $C$ is called trivial; otherwise $C$ is non-trivial.

Next, let $S = \{T_1, T_2, \ldots, T_k\}$ be a set of trees satisfying $\Lambda(T_1) = \Lambda(T_2) = \cdots = \Lambda(T_k) = L$ for some leaf label set $L$. A consensus tree for $S$ is a tree that summarizes the branching information contained in $S$ according to some well-defined rule. This paper focuses on the following three variants:

- A cluster that occurs in more than $k/2$ of the trees in $S$ is a majority cluster. A majority rule consensus tree of $S$ [16] is a tree $T$ such that $\Lambda(T) = L$ and $C(T)$ consists of all majority clusters.

- A loose consensus tree of $S$ [4] is a tree $T$ such that $\Lambda(T) = L$ and $C(T)$ consists of all clusters that occur in at least one tree in $S$ and that are compatible with all trees in $S$.

- Let $X'$ be a list of all clusters that occur in at least one tree in $S$, sorted according to the number of trees in $S$ in which they occur (frequently occurring clusters coming first and with ties broken arbitrarily). Construct a set $Y$ of clusters as follows: Initialize $Y := \emptyset$. Then, traverse the list $X'$ and for each cluster $C$ encountered in this order, check if $C$ and $C'$ are pairwise compatible for all $C' \in Y$; if yes then let $Y := Y \cup \{C\}$. A greedy consensus tree of $S$ [5, 10] is a tree $T$ such that $\Lambda(T) = L$ and $C(T) = Y$.

See Figure 1 for an example. As pointed out in [5], for any given $S$, there exists a unique majority rule consensus tree of $S$ and a unique loose consensus tree.

Figure 1: An example. Let $S = \{T_1, T_2, T_3\}$ as shown above with $\Lambda(T_1) = \Lambda(T_2) = \Lambda(T_3) = \{a, b, c, d, e\}$. Majority rule, loose, and greedy consensus trees of $S$ are displayed. Observe that the only non-trivial majority cluster in $S$ is $\{a, b, c\}$. Also observe that $\{d, e\}$ is the only non-trivial cluster in $S$ that is compatible with all trees in $S$. Finally, three different greedy consensus trees of $S$ exist because each of the clusters $\{a, b\}$, $\{a, c\}$, $\{b, c\}$ occurs once in $S$ and exactly one of them will be included in any greedy consensus tree, depending on how ties among clusters are broken.
of \( S \), but a greedy consensus tree of \( S \) is not always uniquely defined. Moreover, if a cluster \( C \) occurs in the majority rule consensus tree of \( S \) or in the loose consensus tree of \( S \), then \( C \) occurs in all greedy consensus trees of \( S \).

1.2 New results. We present fast deterministic algorithms for computing the majority rule consensus tree, the loose consensus tree, and a greedy consensus tree in Sections 3, 4, and 5, respectively, for an input set \( S \) of trees with identical leaf label sets.

The worst-case running times of the previously fastest deterministic algorithms and our new ones are compared below. To express their time complexities, define \( k = |S| \) and \( n = |L| \), and write \( S = \{T_1, T_2, \ldots, T_k\} \), where \( \Lambda(T_1) = \Lambda(T_2) = \cdots = \Lambda(T_k) = L \). Observe that \( n + 1 \leq |V(T_i)| \leq 2n - 1 \) for every \( i \in \{1, 2, \ldots, k\} \). Let \( p \) be the number of different clusters occurring in \( S \) and \( q \) the total number of clusters occurring in \( S \) (with repetitions). Thus, \( p \leq q \) and \( q = \Theta(nk) \) with \( k \cdot (n + 1) \leq q \leq k \cdot (2n - 1) \).

<table>
<thead>
<tr>
<th>Majority rule consensus tree</th>
<th>Previously best: ( O(n^2 + nk^3) ) time (Wareham [27])</th>
</tr>
</thead>
<tbody>
<tr>
<td>This paper:</td>
<td>( O(nk \log k) ) time Section 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loose consensus tree</th>
<th>Previously best: ( O(nq^2) = O(n^3k^2) ) time (McMorris &amp; Wilkinson [18])</th>
</tr>
</thead>
<tbody>
<tr>
<td>This paper:</td>
<td>( O(nk) ) time Section 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Greedy consensus tree</th>
<th>Previously best: ( O(nq + n^2p) = O(n^3k) ) time (Bryant [5])</th>
</tr>
</thead>
<tbody>
<tr>
<td>This paper:</td>
<td>( O(nq) = O(n^2k) ) time Section 5</td>
</tr>
</tbody>
</table>

Note that our algorithm for the loose consensus tree is optimal since the input size is \( \Omega(nk) \).

Also note that a randomized, hashing-based algorithm for constructing the majority rule consensus tree was given by Amenta et al. in [1]; its expected running time is \( O(nk) \) but its worst-case running time is unbounded.

We implemented our algorithms to make sure that they are practical and applied them to various simulated data sets, as explained in Section 6. In short, these experiments suggested that the running times of our deterministic algorithms are already comparable to (and in many cases, better than) those of the methods found in commonly used software packages such as PHYLIP [10], without having to use randomization and hash tables for storing the clusters occurring in \( S \).

2 Preliminaries

This section describes some algorithmic tools that will be used in the paper.

2.1 Day’s algorithm [7]. Day’s algorithm [7] is a method that takes as input two trees \( T_{ref} \) and \( T \) with \( \Lambda(T_{ref}) = \Lambda(T) = L \), and after some preprocessing, can check whether any specified cluster from \( C(T) \) also occurs in \( C(T_{ref}) \) efficiently. In particular, it can be applied to identify the set of all clusters that occur in both \( T_{ref} \) and \( T \) in \( O(n) \) time, where \( n = |L| \). We summarize this useful result as follows:

**Theorem 2.1.** (Day [7]) Let \( T_{ref} \) and \( T \) be two 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 \( \Lambda(T[u]) \in C(T_{ref}) \) in \( O(1) \) time.

2.2 The delete and insert operations. Define the delete operation on any non-root, internal node \( u \) in a tree as the operation of letting all of \( u \)’s children become children of the parent of \( u \), and then removing \( u \) and the edge between \( u \) and its parent. See Figure 2. Importantly, any delete operation on a node \( u \) in a tree \( T \) removes the cluster \( \Lambda(T[u]) \) from the cluster collection \( C(T) \) without affecting the other clusters. Conversely, define the insert operation as the operation that creates a new node \( u \) which becomes: (1) a child of an existing internal node \( v \), and (2) the parent of a proper subset \( X \) of \( v \)'s children satisfying \( |X| \geq 2 \); as a consequence, a new cluster \( \Lambda(T[u]) = \bigcup_{v_i \in X} \Lambda(T[v_i]) \) is inserted into \( C(T) \).

2.3 Checking compatibility. Suppose that a tree \( T \) is given. For any cluster \( C \subseteq \Lambda(T) \), let \( Child(C) \) be the set of children of the node \( lca^T(C) \). The next lemma characterizes when \( C \) is compatible with \( T \).

**Lemma 2.1.** For any tree \( T \) and \( C \subseteq \Lambda(T) \), \( C \) is compatible with \( T \) if and only if \( |C \cap \Lambda(T[c_i])| \) equals 0 or \( |\Lambda(T[c_i])| \) for each \( c_i \in Child(C) \).

**Proof.** (\( \rightarrow \)) We prove the contrapositive. Suppose there exists a \( c_i \in Child(C) \) with \( 0 < |C \cap \Lambda(T[c_i])| < |\Lambda(T[c_i])| \). This implies that \( \Lambda(T[c_i]) \) contains some element \( x \in C \) and some element \( y \notin C \). Since \( c_i \) is a child of \( lca^T(C) \), there exists an element \( z \in C \) which
is a descendant of another child $c_j$ in $\text{Child}(C)$, i.e., $z \not\in \Lambda(T[c_i])$. But then $(x, z) \subseteq C$ and $y \not\in C$, while $(x, y) \subseteq \Lambda(T[c_i])$ and $z \not\in \Lambda(T[c_i])$, so $C$ and $\Lambda(T[c_i])$ are not pairwise compatible. By definition, $C$ is not compatible with $T$.

(\neg) Consider any $u \in V(T)$. There are three cases:

- $u$ is an ancestor of $\text{lea}^T(C)$: Then trivially $C \subseteq \Lambda(T[u])$.
- $u$ is a descendant of $\text{lea}^T(C)$: Let $c_i$ be the child of $\text{lea}^T(C)$ which is an ancestor of $u$. By the lemma statement, $|C \cap \Lambda(T[c_i])|$ equals either 0 or $|\Lambda(T[c_i])|$. If the former holds then $C \cap \Lambda(T[u]) = \emptyset$; if the latter holds then $C \cap \Lambda(T[u]) = \Lambda(T[u])$, and $\Lambda(T[u]) \subseteq C$.
- $u$ is not an ancestor and not a descendant of $\text{lea}^T(C)$: Then $C \cap \Lambda(T[u]) = \emptyset$.

In all cases, $C$ and $\Lambda(T[u])$ are pairwise compatible. Thus, $C$ is compatible with $T$. \hfill \square

2.4 Procedure $\text{Merge}_\text{Trees}$. Let $T_1$ and $T_2$ be two trees with $\Lambda(T_1) = \Lambda(T_2) = L$ such that every cluster in $\mathcal{C}(T_1)$ is compatible with $T_2$. (Note that this condition is equivalent to requiring that every cluster in $\mathcal{C}(T_2)$ is compatible with $T_1$.) This subsection describes an $O(n)$-time procedure $\text{Merge}_\text{Trees}(T_1, T_2)$ which returns a tree $T$ with $\Lambda(T) = L$ and $\mathcal{C}(T) = \mathcal{C}(T_1) \cup \mathcal{C}(T_2)$, where $n = |L|$. Hence, $\text{Merge}_\text{Trees}$ combines all the clusters from two non-conflicting trees into one tree in linear time.

The procedure operates in two phases. The first phase is a preprocessing phase which works as follows. As in Day’s algorithm [7] (mentioned above), do an $O(n)$-time depth-first traversal of $T_1$ to construct a leaf numbering function $f$, i.e., a bijection from $L$ to the set $\{1, 2, \ldots, n\}$, under which each $C \in \mathcal{C}(T_1)$ is represented by an interval of consecutive integers. Then, relabel all leaves in $T_2$ according to $f$, and do a bottom-up traversal of $T_2$ to obtain and store, for each $v \in V(T_2)$, the value $m(v) := \min_{x \in \Lambda(T[v])} \{f(x)\}$. Also do a single top-down traversal of $T_2$ to compute, for each $v \in V(T_2)$, the number of edges from the root of $T_2$ to $v$ and store it in $\text{depth}(v)$. Transform $T_2$ into an ordered tree by ordering the children at each internal node $v$ of $T_2$ so that for every two children $a$ and $b$ of $v$, $a$ is to the left of $b$ if and only if $m(a) < m(b)$. Now Lemma 2.1 implies:

**Lemma 2.2.** After making $T_2$ an ordered tree as described above, any $C \subseteq L$ is compatible with $T_2$ if and only if $C$ is of the form $C = \bigcup_{c \in D} \Lambda(T_2[c_i])$, where $D$ is a consecutive subsequence of the children of the node $\text{lea}^T(C)$.

Therefore, when inserting a cluster of the form $\Lambda(T_1[u])$ into $T_2$, we have to create a new child node $c$ of the node $r_u := \text{lea}^T(C(T_1[u]))$ and let a consecutive subsequence of the children of $r_u$ become children of $c$ instead. To be able to identify this consecutive subsequence of children, we need to find the leftmost and rightmost children of $r_u$ whose subtrees contain leaves from $\Lambda(T_1[u])$. For this purpose, for each $x \in L$, first define $\text{leaf\_rank}_{T_2}(x)$ to be $1 + \text{(the number of leaves to the left of } x \text{ in } T_2)$. Then, for every $u \in V(T_1)$, define $\text{start}(u) := \min_{x \in \Lambda(T_1[u])} \text{leaf\_rank}_{T_2}(x)$ and $\text{stop}(u) := \max_{x \in \Lambda(T_1[u])} \text{leaf\_rank}_{T_2}(x)$. Intuitively, $\text{start}(u)$ and $\text{stop}(u)$ tell us the interval in the left-to-right ordering of the leaves in $T_2$ that consists of all leaves from $\Lambda(T_1[u])$. Use the following recursive formula to precompute $\text{start}(u)$ and $\text{stop}(u)$ for all $u \in V(T_1)$ in $O(n)$ time in total:

**Lemma 2.3.** For any $u \in V(T_1)$, let $\text{Child}(u)$ be the set of children of $u$. Then:

$$\begin{align*}
\text{start}(u) &= \begin{cases} 
\text{leaf\_rank}_{T_2}(u), & \text{if } u \text{ is a leaf} \\
\min_{c \in \text{Child}(u)} \text{start}(c), & \text{otherwise}
\end{cases} \\
\text{stop}(u) &= \begin{cases} 
\text{leaf\_rank}_{T_2}(u), & \text{if } u \text{ is a leaf} \\
\max_{c \in \text{Child}(u)} \text{stop}(c), & \text{otherwise}
\end{cases}
\end{align*}$$

Next, for every $x \in L$, define $x_{\text{left}}$ as the node $v$ in $T_2$ with the smallest value of $\text{depth}(v)$ (i.e., as close to the root as possible) whose leftmost leaf descendant is $x$. Define $x_{\text{right}}$ for every $x \in L$ analogously, but using the rightmost leaf descendant instead. See Figure 3. To compute $x_{\text{left}}$ and $x_{\text{right}}$ for all $x \in L$, do an $O(n)$-time bottom-up traversal of $T_2$. Finally, apply the method of [3] to preprocess $T_2$ in $O(n)$ time so that every
subsequent lca-query on any two nodes in \( T_2 \) can be answered in \( O(1) \) time. This concludes the first phase.

We now describe the second phase of \textsc{Merge.Trees} which inserts clusters from \( T_1 \) into \( T_2 \). (Recall from the first paragraph in this subsection that \textsc{Merge.Trees} requires every \( \Lambda(T_1[u]) \) to be compatible with \( T_2 \).) To avoid changing the parent of any node in \( T_2 \) more than once, we use a bottom-up approach. For each \( u \in V(T_1) \) in bottom-up order, do the following steps: Retrieve \( a := \text{start}(u) \) and \( b := \text{stop}(u) \), and let \( a' \) and \( b' \) be the elements of \( L \) satisfying \( \text{leaf\_rank}(a') = a \) and \( \text{leaf\_rank}(b') = b \). Obtain \( r_u := \text{lca}(\{a', b'\}) \) in \( O(1) \) time by querying the lca data structure. The next lemma tells us where in \( T_2 \) to insert \( \Lambda(T_1[u]) \) as a new cluster. See Figure 3 for an illustration.

\textbf{Lemma 2.4.} Let \( d_u \) be the leftmost child of \( r_u \) such that \( \Lambda(T_1[u]) \cap \Lambda(T_2[d_u]) \neq \emptyset \), and \( e_u \) the rightmost child of \( r_u \) such that \( \Lambda(T_1[u]) \cap \Lambda(T_2[e_u]) \neq \emptyset \). The following holds:

1. If \( \text{depth}(a'_\text{left}) > \text{depth}(r_u) \) then \( d_u = a'_\text{left} \); otherwise, \( d_u = \text{the leftmost child of } r_u \).
2. If \( \text{depth}(b'_\text{right}) > \text{depth}(r_u) \) then \( e_u = b'_\text{right} \); otherwise, \( e_u = \text{the rightmost child of } r_u \).

Thus, apply Lemma 2.4 to find \( d_u \) and \( e_u \) in \( O(1) \) time. In case \( d_u \) is the leftmost child of \( r_u \) and \( e_u \) is the rightmost child of \( r_u \) then \( \Lambda(T_2[r_u]) = \Lambda(T_1[u]) \), i.e., the cluster already occurs in \( T_2 \) and we do nothing. Otherwise, create a new child \( c \) of \( r_u \), set \( \text{depth}(c) := \text{depth}(r_u) + 1 \), and let all children of \( r_u \) in the sequence \( d_u, \ldots, e_u \) become children of \( c \). Update \( a'_\text{left} \) to point to \( c \) if \( d_u \) was not the leftmost child of \( r_u \), and update \( b'_\text{right} \) analogously. The correctness follows from Lemma 2.2.

In the second phase, since the nodes are treated in bottom-up order, the parent of each node in \( T_2 \) changes at most once. Furthermore, due to the bottom-up ordering, there is no need to update any depth-values or lca-values for nodes in \( T_2 \) although they will change during execution. For each \( u \in V(T_1) \), we perform \( O(1) \) additional operations. In total, everything takes \( O(n) \) time.

\textbf{Theorem 2.2.} Procedure \textsc{Merge.Trees}(\( T_1, T_2 \)) runs in \( O(n) \) time, where \( n = |L| \).

\section{Constructing the majority rule consensus tree}

Here, we present a recursive algorithm named \textsc{Maj.Rule.Cons.Tree} for constructing the majority rule consensus tree of \( S \) in \( O(nk \log k) \) time. The pseudocode is shown in Figure 4.

\textsc{Maj.Rule.Cons.Tree} splits the input set \( S \) of trees into two halves, recursively constructs the majority rule consensus trees \( T^A \) and \( T^B \) for each of the halves, checks every cluster that occurs in \( T^A \) or \( T^B \) and deletes it if it is not a majority cluster of \( S \), and finally builds the majority rule consensus tree of \( S \) by combining the resulting \( T^A \) and \( T^B \). To combine \( T^A \) and \( T^B \) in the last step, it applies the procedure \textsc{Merge.Trees} from Section 2.4.

To prove the correctness of the above approach, we need the following lemma:

\textbf{Lemma 3.1.} Let \( T^A \) and \( T^B \) be the majority rule consensus trees of \( S^A = \{T_1, \ldots, T_{(k/2)}\} \) and \( S^B = \{T_{(k/2)+1}, \ldots, T_k\} \), respectively. Every majority cluster of \( S \) occurs in at least one of \( T^A \) and \( T^B \).

\textbf{Proof.} Consider any majority cluster \( C \) of \( S \). By definition, \( C \) occurs in strictly more than \( k/2 \) of the trees in \( S \). First suppose that \( k \) is even. Then \( C \) occurs in at least \( \frac{k}{2} + 1 \) trees from \( S \), and at least half of these trees belong to either \( S^A \) or \( S^B \). Since \( (\frac{k}{2} + 1)/2 > (\frac{k}{2})/2 \) and \( |S^A| = |S^B| = k/2 \), we have \( C \in C(T^A) \) or \( C \in C(T^B) \) by the definition of the majority rule consensus tree.

On the other hand, if \( k \) is odd then \( C \) occurs in at least \( \frac{k+1}{2} \) trees from \( S \). Moreover, when \( k \) is odd, \( |S^A| = (k+1)/2 \) and \( |S^B| = (k-1)/2 \). For the sake of obtaining a contradiction, suppose that \( C \) occurs in at most \( \frac{k-1}{2} \) trees in \( S^A \) and in at most \( \frac{k-1}{2} \) trees in \( S^B \). Then the total number of occurrences of \( C \) in \( S \)
Algorithm Maj_Rule_Cons_Tree
Input: A set $S = \{T_1, T_2, \ldots, T_k\}$ of trees with $\Lambda(T_i) = \Lambda(T_2) = \cdots = \Lambda(T_k)$.  
Output: The majority rule consensus tree of $S$.

1 if $k = 1$ then let $T := T_1$.  
   /* Base case of the recursion */
2 else
   2.1 Let $T^A := \text{Maj}_\text{Rule}_\text{Cons}_\text{Tree}(T_1, \ldots, T_{k/2})$, $T^{B} := \text{Maj}_\text{Rule}_\text{Cons}_\text{Tree}(T_{k/2+1}, \ldots, T_k)$.
   2.2 for each $v \in V(T^A)$ in top-down order do
      if $\Lambda(T^A[v])$ occurs in at most $k/2$ trees in $S$ then
         do a delete operation on node $v$.
   2.3 for each $v \in V(T^B)$ in top-down order do
      if $\Lambda(T^B[v])$ occurs in at most $k/2$ trees in $S$ then
         do a delete operation on node $v$.
   2.4 Let $T := \text{Merge}_\text{Trees}(T^A, T^B)$.
3 return $T$.  
End Maj_Rule_Cons_Tree

Figure 4: Algorithm Maj_Rule_Cons_Tree for constructing the majority rule consensus tree.

is at most $\frac{k+1}{4} + \frac{k-1}{4} = \frac{k}{2}$. Contradiction. Therefore, $C$ occurs in strictly more than $\frac{k+1}{4} = \frac{(k+1)/2}{2}$ trees in $S^A$ (in which case $C \in C(T^A)$ by the definition of the majority rule consensus tree) or in strictly more than $\frac{k-1}{4} = \frac{(k-1)/2}{2}$ trees in $S^B$ (in which case $C \in C(T^B)$).

This yields:

**Theorem 3.1.** Algorithm Maj_Rule_Cons_Tree constructs the majority rule consensus tree of $S$ in $O(nk \log k)$ time.

**Proof.** According to Lemma 3.1, every majority cluster of $S$ belongs to $C(T^A) \cup C(T^B)$. Thus, after checking every cluster $C$ that occurs in $T^A$ or $T^B$ and keeping $C$ if and only if it occurs in more than $k/2$ of the trees in $S$ in Steps 2.2 and 2.3, the resulting $C(T^A) \cup C(T^B)$ will be equal to the set $M$ of majority clusters of $S$. All clusters in $M$ are pairwise compatible because any pair of clusters in $M$ must occur together in at least one tree in $S$ by the pigeonhole principle, so the procedure Merge_Trees$(T^A, T^B)$ will produce a tree $T$ with $C(T) = M$ in Step 2.4. By definition, $T$ is the majority rule consensus tree of $S$. This proves the correctness of Maj_Rule_Cons_Tree.

Next, we describe how to implement Step 2.2 efficiently. Augment each internal node $v$ in $T^A$ with a counter $count(v)$ and initialize $count(v) := 0$. For each $T_i \in S$, apply the preprocessing of Day’s algorithm (see Section 2.1) using $T_{ref} := T^A$ and $T := T_i$, and then traverse $T_i$ while checking each of the $O(n)$ internal nodes $u$ of $T_i$ in $O(1)$ time to see if $\Lambda(T_i[u])$ belongs to $C(T^A)$; if yes, then increase $count(v)$ for the corresponding internal node in $T^A$ by one. Due to Theorem 2.1, this takes $O(nk)$ time in total. Then, remove the non-majority clusters from $C(T^A)$ by traversing $T^A$ and performing a delete operation on every internal node $v$ in $T^A$ with $count(v) \leq k/2$. In total, all delete operations take $O(n)$ time because the nodes are considered in top-down order (whenever a node is deleted, its children are moved but they will not be moved again later), and the sum of the number of children taken over all nodes is $O(n)$. Proceed analogously for $T^B$ in Step 2.3. In summary, Steps 2.2 and 2.3 take $O(nk)$ time. Lastly, Step 2.4 takes $O(n)$ time according to Theorem 2.2.

Let $t(k)$ be the running time of Maj_Rule_Cons_Tree$(T_1, T_2, \ldots, T_k)$. Then $t(k) = 2 \cdot t(k/2) + O(nk)$ with $t(1) = O(n)$. Solving the recurrence relation gives $t(k) = O(nk \log k)$.

**Remark:** A natural way to parameterize the majority rule consensus tree is by letting $\ell$ be any real number such that $1/2 \leq \ell \leq 1$, and taking only clusters that occur in more than a fraction $\ell$ of the input trees in $S$ [17]. Algorithm Maj_Rule_Cons_Tree can be modified accordingly without affecting the time complexity by changing the condition “in at most $k/2$” in Steps 2.2 and 2.3.

4 Constructing the loose consensus tree

The loose consensus tree of $S$ can be computed by testing every cluster that occurs in $S$ against all other clusters in $S$ for compatibility. Since each pair of clusters can be checked in $O(n)$ time, this yields an algorithm with $O(nq^2) = O(n^3 k^2)$ running time. (If we incorporate a bottom-up technique based on Lemma 2.1 to check a cluster for compatibility with a tree in $O(n)$ time, the running time is slightly improved to $O(nkq) = O(n^3 k^2)$.) Below, we show how to do it in $O(nk)$ time, which is optimal. Our algorithm is called Loose_Cons_Tree. It uses Merge_Trees from Section 2.4 as well as a procedure named One-Way_Compatible as subroutines. We first describe One-Way_Compatible.

4.1 Procedure One-Way_Compatible. Let $T_1$ and $T_2$ be two trees with $\Lambda(T_1) = \Lambda(T_2) = L$. Procedure One-Way_Compatible$(T_1, T_2)$ outputs a tree $T$ with $\Lambda(T) = L$ such that $C(T) = \{C \in C(T_1) : C$ is compatible with $T_2\}$. In other words, One_Way_Compatible$(T_1, T_2)$ returns a copy of $T_1$ in which every cluster that is not compatible with $T_2$ has been removed. The procedure is asymmetric; for exam-
ple, 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($T_2, T_1) = T_2$.

Procedure One-Way Compatible is similar to Merge_Trees in Section 2.4. It also operates in two phases, where the first phase is a preprocessing phase and the second phase traverses $T_1$. The first phase of One-Way Compatible performs all the steps of the first phase of Merge_Trees, plus a bottom-up traversal of $T_1$ to obtain and store, for every $u \in V(T_1)$, the value size($u$) := |$\Lambda(T_1[u])$|.

The second phase of One-Way Compatible differs from that of Merge_Trees. Instead of inserting new nodes into $T_2$, it deletes all nodes from $T_1$ whose associated clusters are not compatible with $T_2$. To check if $\Lambda(T_1[u])$ for any $u \in V(T_1)$ is compatible with $T_2$ in $O(1)$ time, apply the following technique (refer to Section 2.4 for explanations of the notation used below). Assign $a := \text{start}(u)$ and $b := \text{stop}(u)$, and let $a'$ and $b'$ be the elements of $L$ such that leaf_rank($a'$) = $a$ and leaf_rank($b'$) = $b$. Compute $r_u := \text{lca}(T_2([a', b']))$ in $O(1)$ time by querying the lca data structure. Next, if depth($a'_c$) $>$ depth($r_u$) then define $d_u := a'_c$; otherwise, define $d_u :=$ the leftmost child of $r_u$. Similarly, if depth($b'_c$) $>$ depth($r_u$) then define $e_u := b'_c$; otherwise, define $e_u :=$ the rightmost child of $r_u$. The value $|\Lambda(T_1[u])|$ is retrieved from size($u$) in $O(1)$ time. Then:

**Lemma 4.1.** $\Lambda(T_1[u])$ is compatible with $T_2$ if and only if: (i) the parent of $d_u$ is $r_u$; (ii) the parent of $e_u$ is $r_u$; and (iii) $|\Lambda(T_1[u])| = b - a + 1$.

**Proof.** Let $C$ denote the cluster $\Lambda(T_1[u])$. Lemma 2.2 states that $C$ is compatible with $T_2$ if and only if $C = \Lambda(T_2[c_1]) \cup \cdots \cup \Lambda(T_2[c_j])$ for some consecutive subsequence $c_1, c_{i+1}, \ldots, c_j$ of the children of the node $r_u$.

($\rightarrow$) Suppose $C = \Lambda(T_2[c_1]) \cup \cdots \cup \Lambda(T_2[c_j])$, where $c_1, \ldots, c_j$ is a consecutive subsequence of children of $r_u$.

If $i = 1$ then $d_u$ is the leftmost child of $r_u$ by definition. If $i > 1$ then $a'$ is the leftmost leaf in $T_2[c_i]$ and $d_u = a'_{\text{left}}$ must be child of $r_u$ because otherwise, there would exist some other leaf from $C$ to the left of $a'$, which is impossible. Therefore, $d_u$ is always a child of $r_u$, and we have (i). An analogous argument shows that (ii) also holds. To prove (iii), note that the $|C|$ elements of $C$ occur as a consecutive block starting at position $a$ and ending at position $b$ in the left-to-right ordering of the leaves in $T_2$, which means that $|C| = b - a + 1$.

($\leftarrow$) Suppose (i), (ii), and (iii) hold. By the definition of $a'_{\text{left}}$, the leftmost leaf descendant of every node on the path in $T_2$ between $a'$ and $a'_{\text{left}}$ is $a'$. Thus, the leftmost leaf descendant of $d_u$ is $a'$. In the same way, the rightmost leaf descendant of $e_u$ is $b'$. Then, conditions (i) and (ii) imply that $C \subseteq \Lambda(T_2[d_u]) \cup \cdots \cup \Lambda(T_2[e_u])$, where $d_u, \ldots, e_u$ is a consecutive subsequence of children of $r_u$. There are exactly $b - a + 1$ leaves in the interval $a'..b'$ in the left-to-right ordering of $T_2$, so $|\Lambda(T_2[d_u]) \cup \cdots \cup \Lambda(T_2[e_u])| = b - a + 1$. From condition (iii), $|C| = |\Lambda(T_2[d_u]) \cup \cdots \cup \Lambda(T_2[e_u])|$, which shows that $C = \Lambda(T_2[d_u]) \cup \cdots \cup \Lambda(T_2[e_u])$, where $d_u, \ldots, e_u$ is a consecutive subsequence of children of $r_u$.

Now, the second phase of One-Way Compatible is: For each $u \in V(T_1)$, apply Lemma 4.1 and if $\Lambda(T_1[u])$ is compatible with $T_2$ then mark $u$ as “good”; otherwise, mark $u$ as “bad”. Next, traverse $T_1$ in top-down order and for each node $u \in V(T_1)$ encountered, if $u$ is “bad” then perform a delete operation on $u$.

In total, the first phase takes $O(n)$ time. The time complexity of the second phase is $O(n)$ because each compatibility check takes $O(1)$ time by applying Lemma 4.1. Also, the total time needed for all node deletions is $O(n)$ since whenever a node $u$ in $T_1$ is deleted, the children of $u$ are moved but those nodes will never need to be moved again because of the top-down order. Hence, the contribution to the total running time of each node is (at most) proportional to the number of children it has, and the sum of the number of children taken over all nodes in $T_1$ is $O(n)$.

**Theorem 4.1.** Procedure One-Way Compatible($T_1, T_2$) runs in $O(n)$ time, where $n = |L|$.

**4.2 Algorithm Loose_Cons_Tree.** First, for any $j \in \{1, \ldots, k\}$, we define the set of one-way compatible clusters up to $j$ as the set $O_j = \bigcup_{i=1}^{j} (C \in C(T_i) : C \text{ is compatible with all trees in } \{T_i, T_{i+1}, \ldots, T_j\})$. It is easy to see that:

**Lemma 4.2.** For any $j \in \{1, \ldots, k\}$, all clusters in $O_j$ are pairwise compatible.

**Proof.** Consider any two clusters $C, C' \in O_j$. If $j = 1$ or if $C$ and $C'$ occur in the same tree $T_i$ then the lemma is trivially true. Therefore, assume without loss of generality that $j \geq 2$ and $C \subseteq C(T_i)$ and $C' \subseteq C(T_i')$, where $i < i' \leq j$. Since $C \in O_j$, $C$ is compatible with all trees in $\{T_i, \ldots, T_j\}$ and thus compatible with $T_{i'}$. This means that $C$ and $C'$ are pairwise compatible.

Then, according to Theorem 3.5.2 in [22], the set $O_j$ equals the cluster collection of a uniquely defined tree for each $j \in \{1, \ldots, k\}$. Define $R_j$ to be the tree...
with \( C(R_j) = \mathcal{O}_j \). Clearly, \( R_1 = T_1 \). To obtain \( R_j \) for any \( j \in \{2, \ldots, k\} \), we shall use the following recursive formulation:

**Lemma 4.3.** Let \( j \in \{2, \ldots, k\} \) and \( A = \text{One-Way-Compatible}(R_{j-1}, T_j) \). Then \( \text{Merge_Trees}(A, T_j) \) is equal to the tree \( R_j \).

**Proof.** By definition, \( C(R_{j-1}) = \mathcal{O}_{j-1} \) and \( A = \text{One-Way-Compatible}(R_{j-1}, T_j) \) is a tree whose cluster collection \( C(A) \) is the subset of \( C(R_{j-1}) \) consisting of those clusters that are also compatible with \( T_j \). Thus, \( C(A) = \bigcup_{i=1}^{j-1} \{ C \in C(T_i) : C \text{ is compatible with all trees in } \{T_1, \ldots, T_j\} \} \). Consequently, \( \text{Merge_Trees}(A, T_j) \) returns a tree whose cluster collection is equal to \( C(A) \cup C(T_j) \). Trivially, all clusters occurring in \( T_j \) are compatible with \( T_j \), so \( C(A) \cup C(T_j) = \bigcup_{i=1}^{j} \{ C \in C(T_i) : C \text{ is compatible with all trees in } \{T_1, \ldots, T_j\} \} = \mathcal{O}_j \). Hence, this tree is equal to \( R_j \). \( \square \)

Next, we show that \( C(T) \subseteq C(R_k) \), where \( T \) is the loose consensus tree of \( S \).

**Lemma 4.4.** Let \( T \) be the loose consensus tree of \( S \). Every cluster that occurs in \( T \) also occurs in \( R_k \).

**Proof.** Let \( C \) be any cluster in \( C(T) \). By the definition of the loose consensus tree, \( C \in C(T_j) \) for some \( j \in \{1, \ldots, k\} \) and \( C \) is compatible with all trees in \( \{T_1, \ldots, T_k\} \). In particular, \( C \) is compatible with the trees \( \{T_j, \ldots, T_k\} \), so \( C \in \mathcal{O}_k \), i.e., \( C \) occurs in \( R_k \). \( \square \)

As suggested by Lemma 4.4, one strategy for computing the loose consensus tree of \( S \) is to build the tree \( R_k \) and then remove certain clusters from it. The next lemma tells us which ones.

**Lemma 4.5.** Let \( T \) be the loose consensus tree of \( S \). Then \( C(T) = \{ C \in C(R_k) : C \text{ is compatible with all trees in } \{T_1, \ldots, T_k\} \} \).

**Proof.** Consider any \( C \in C(R_k) \). Then for some \( i \in \{1, \ldots, k\} \), \( C \in C(T_i) \) and \( C \) is compatible with all trees in \( \{T_1, \ldots, T_k\} \). If \( C \) is also compatible with all trees in \( \{T_1, \ldots, T_k\} \) then \( C \) belongs to the set \( \{ C \in \bigcup_{i=1}^{k} C(T_i) : C \text{ is compatible with all trees in } \{T_1, \ldots, T_k\} \} \), which is equal to \( C(T) \) by the definition of the loose consensus tree.

For the other direction, consider any \( C \in C(T) \). Then \( C \in C(R_k) \) by Lemma 4.4, and \( C \) is compatible with all trees in \( \{T_1, \ldots, T_k\} \) by the definition of the loose consensus tree. \( \square \)

Algorithm **Loose_Cons_Tree** is shown in Figure 5. Its correctness follows from Lemmas 4.4 and 4.5. To analyze its time complexity, observe that every execution of **One-Way-Compatible** takes \( O(n) \) time according to Theorem 4.1 and every execution of **Merge_Trees** takes \( O(n) \) time by Theorem 2.2, so Step 2 takes \( O(nk) \) time. For the same reason, Step 4 takes \( O(nk) \) time. We have:

**Theorem 4.2.** Algorithm **Loose_Cons_Tree** constructs the loose consensus tree of \( S \) in \( O(nk) \) time.

5 Constructing a greedy consensus tree

We now give an algorithm for building a greedy consensus tree of \( S \) in \( O(nq) = O(n^3k) \) time. Recall that \( p \) is the number of different clusters and \( q \) the total number of clusters occurring in \( S \), with repetitions.

A straightforward implementation of the method outlined in Section 2.1.4 in [5] (summarized in Section 1.1 above) leads to a time complexity of \( O(nq + n^3p) = O(n^3k) \). Our improvement comes from eliminating one of the bottlenecks: Instead of first building a maximal set \( \mathcal{V} \) of pairwise compatible clusters in \( O(n^3p) \) time and then constructing a tree \( T \) from \( \mathcal{V} \), we build \( T \) directly by inserting one cluster at a time, using an \( O(n) \)-time method made possible by Theorem 4.2.

**Lemma 5.1.** For any tree \( T \) and \( C \subseteq \Lambda(T) \) with \( C \not\in C(T) \), it is possible to determine if \( C \) is compatible with \( T \), and if so, insert \( C \) into \( C(T) \) in \( O(n) \) time, where \( n = |\Lambda(T)| \).

**Proof.** Create a tree \( T' \) with \( \Lambda(T') = \Lambda(T) \) in which all leaves belonging to \( C \) have a common parent node attached to the root of \( T' \) and all leaves in \( \Lambda(T') \setminus C \) are attached to the root directly. Clearly, the only non-trivial cluster occurring in \( T' \) is \( C \). Let \( T_{\text{loose}} \) be the loose consensus tree of \( \{T, T'\} \). By definition, \( C \) is compatible with \( T \) if and only if \( C(T_{\text{loose}}) = \)
The total time complexity is therefore \(O(nq)\) time by using radix sort to sort \(W\). Step 3 sorts \(p\) integers belonging to \(\{1, 2, \ldots, k\}\), which takes \(O(k+p)\) time with counting sort. Step 4 uses \(O(n)\) time, and Step 5 takes \(O(np)\) time because of Lemma 5.1. The total time complexity is therefore \(O(nq+k+p+n+np) = O(nq)\).

**Theorem 5.1.** Algorithm Greedy_Cons_Tree constructs a greedy consensus tree of \(S\) in \(O(nq)\) time.

**6 Implementations and experimental results**

We implemented our algorithms for constructing majority rule, loose, and greedy consensus trees in the C++ programming language. Section 6.1 below describes a number of modifications that were made to obtain fast running times in practice. Observe that the modified algorithms achieve the same worst-case time complexities as in Sections 3–5 and remain fully deterministic. Specifically, we still do not use randomization and hash tables for storing clusters.

After implementing the algorithms, we ran them on simulated data sets of varying sizes and compared their running times to those of some freely available, widely used software: PHYLIP [10], SunTrees in DendroPy [24], and COMPONENT [20]. (We did not compare our methods to PAUP* [26] because it is commercial software which we did not have access to.) The results are reported in Section 6.2.

We have combined our prototype implementations into a package which we call FACT (Fast Algorithms for Consensus Trees). A web interface to FACT has been set up at:

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

The source code of FACT may also be obtained from there, or directly from the authors.

**6.1 Fast implementations of our algorithms**

**Majority rule consensus tree:** Algorithm Maj_Rule_Cons_Tree in Section 3 uses recursion to build the majority rule consensus tree. To speed up its implementation, we eliminate some of the overhead for small instances by breaking the recursion at \(k = 2\) and switching over to a naive method at this point, instead of letting the recursion run all the way down to \(k = 1\).

**Majority rule consensus tree and loose consensus tree:** A special data structure that can answer lca-queries in \(O(1)\) time after linear-time preprocessing [3] was used in the descriptions of the procedures Merge_Trees in Section 2.4 and One-Way_Compatible in Section 4.1. Although this leads to conceptually simple and asymptotically optimal algorithms, the linear-time preprocessing has a high constant factor. A faster (and more easily codable) alternative that does not need [3] is presented below.

We use the same definitions as in Sections 2.4 and 4.1. For any node \(u \in V(T_i)\), let \(a := \text{start}(u)\), \(b := \text{stop}(u)\), and let \(a'\) and \(b'\) be leaves in \(L\) such that \(\text{leaf}\_\text{rank}(a') = a\) and \(\text{leaf}\_\text{rank}(b') = b\). If we refer back to Lemma 4.1, it seems that the lca is required because we need to check whether the parent of \(d_u\) and the parent of \(e_u\) are both equal to \(r_u\). However, we can make use of the correctness of Lemma 4.1 to deduce that \(\Lambda(T_1[u])\) is compatible with \(T_2\) if and only if:

```
Algorithm Greedy_Cons_Tree
Input: A set \(S = \{T_1, T_2, \ldots, T_k\}\) of trees with \(\Lambda(T_1) = \Lambda(T_2) = \cdots = \Lambda(T_k) = L\).
Output: A greedy consensus tree of \(S\).
1. Fix an arbitrary ordering of \(L\). For each \(T_i \in S\), do a bottom-up traversal of \(T_i\) to compute a bit vector of length \(n\) for each node \(u\) that indicates which leaves belong to \(\Lambda(T_i[u])\).
2. Put all of the resulting bit vectors (\(q\) in total) in a list \(\mathcal{W}\), sort \(\mathcal{W}\), and do a single scan of \(\mathcal{W}\) to identify the \(p\) different clusters in \(\mathcal{S}\) and the number of occurrences of each one.
3. Store the different clusters of \(\mathcal{S}\) and their frequencies in a list \(\mathcal{X}\). Sort \(\mathcal{X}\) in order of non-increasing frequency.
4. Let \(T\) be a tree consisting of a root node attached to \(n\) leaves labeled by \(L\).
5. for \(i := 1\) to \(p\) do
   
   If the \(i\)th cluster of the list \(\mathcal{X}\) is non-trivial then try to insert it into \(C(T)\) as described in Lemma 5.1.
6. return \(T\)
End Greedy_Cons_Tree
```

Figure 6: Algorithm Greedy_Cons_Tree for constructing a greedy consensus tree.
• \( \text{depth}(\text{parent}(d_u)) \leq \text{depth}(r_u) \) and \( \text{depth}(\text{parent}(e_u)) \leq \text{depth}(r_u) \).

• The path from \( a' \) to \( \text{parent}(a'_\text{left}) \) and the path from \( b' \) to \( \text{parent}(b'_\text{right}) \) intersect and therefore share at least one common internal node.

• The internal node common to these two paths which has the greatest depth is \( \text{lca}(a', b') \).

We construct and store these paths explicitly during the preprocessing phase. For each leaf \( x \), we store the path from \( x \) to \( x_{\text{left}} \) in \( \text{left}_\text{path}(x) \) and the path from \( x \) to \( x_{\text{right}} \) in \( \text{right}_\text{path}(x) \). By using resizable arrays to store the paths, we can query for a node at a certain depth along any path in \( O(1) \) time.

Given \( a' \) and \( b' \), we assume without loss of generality that \( \text{depth}(a'_{\text{left}}) \geq \text{depth}(b'_{\text{right}}) \). We query \( \text{right}_\text{path}(b') \) for the node on the path from \( b' \) to \( b'_{\text{right}} \) that is at depth \( \text{depth}(a'_{\text{left}}) \). Let \( p_1 := a'_{\text{left}} \) and \( p_2 := \) the corresponding vertex on the path from \( b' \) to \( b'_{\text{right}} \). There are two possibilities:

• If \( p_1 = p_2 \), then \( p_1 \) is the \( \text{lca} \) of \( a' \) and \( b' \), i.e., \( r_u = p_1 \). From this, we deduce that \( d_u \) is the node on \( \text{left}_\text{path}(a') \) at depth \( \text{depth}(a'_{\text{left}}) + 1 \), and \( e_u \) is the node in \( \text{right}_\text{path}(b') \) at the same depth.

• If \( p_1 \neq p_2 \) and \( \text{parent}(p_1) = \text{parent}(p_2) \), then \( \text{parent}(p_1) = \text{lca}(a', b') = r_u \). Therefore, \( p_1 = d_u \) and \( p_2 = e_u \).

After finding \( r_u \), \( d_u \), and \( e_u \) in this way, the procedures \text{Merge}_\text{Trees} \) and \text{One-Way}_\text{Compatible} \) continue their execution as described in Section 2.4 and 4.1.

**Greedy consensus tree:** Step 5 of Algorithm \text{Greedy}_\text{Cons}_\text{Tree} \) in Section 5 checks if given cluster \( C \) is compatible with the current tree \( T \), and if so, inserts it. Lemma 5.1 in Section 5 demonstrated how to do this step in \( O(n) \) time by applying Algorithm \text{Loose}_\text{Cons}_\text{Tree} \) from Section 4. However, since we only need to check a cluster \( C \) (rather than an entire tree), the following direct approach, with the same asymptotic worst-case running time, turns out to be more efficient in practice.

Perform a bottom-up traversal of \( T \) and for each node \( u \in V(T) \), calculate the number of leaves from \( C \) that are in \( \Lambda(T[u]) \). Let \( \text{num}(u) \) denote this number. To compute \( \text{num}(u) \), use the formula \( \text{num}(u) = \sum \text{num}(c_i) \) for every \( c_i \in V(T) \) that is a child of \( u \). The first node \( u \) encountered in the bottom-up traversal that satisfies \( \text{num}(u) = |C| \) is the lowest common ancestor of \( C \) in \( T \). Now, determine if \( C \) is compatible with \( T[u] \) by checking if \( \text{num}(c_i) = |\Lambda(T[c_i])| \) or 0 for every child \( c_i \) of \( u \). This takes \( O(n) \) time and the correctness follows from Lemma 2.1.

If \( C \) is compatible with \( T \) then insert it as follows: Let \( u = \text{lca}_T(C) \) be the node found during the bottom-up traversal described above. Create a new node \( v \), let \( v \) be a child of \( u \), let every child \( c_i \) of \( u \) satisfying \( \text{num}(c_i) = |\Lambda(T[c_i])| \) become a child of \( v \) instead, and return the modified \( T \). Since we change the parent-child relationship of each node at most once, the time complexity of this procedure is also \( O(n) \).

**Constant optimizations:** The computationally most intensive part of \text{Greedy}_\text{Cons}_\text{Tree} \) is the enumeration and counting of clusters in Step 2. Clusters are represented as bit vectors of length \( n \), so to speed up the operations on clusters, we use words of length \( \ell \) to compress each bit vector into \( \lceil \frac{n}{\ell} \rceil \) words. Then, any two clusters can compared in \( O(\frac{n}{\ell}) \) time, allowing the enumeration and counting of clusters in Step 2 to be done in \( O(\frac{np}{\ell}) = O(\frac{n^2}{\ell}) \) time.

**6.2 Experimental results**

**Simulated data sets:** For certain specified values of \( n \) and \( k \), we generated a data set as follows. First, a random tree \( T \) with \( n \) distinctly labeled leaves was created. Here, \( T \) would represent a “true” underlying phylogenetic tree. Next, a set \( S \) of \( k \) conflicting trees with the same leaf label sets was derived from \( T \) by applying random mutations to \( k \) copies of \( T \). Two kinds of mutations were used:

• Delete an internal node \( v \), and attach the children of \( v \) to the parent of \( v \).

• Disconnect a node \( v \), and reattach it to some ancestor of the parent of \( v \).

Before and after each mutation, the following invariant was maintained:

Every internal node has at least two children, and no leaf has any children.

**The methods:** We evaluated the nine different methods listed below. As before, \( n \) is the number of leaves, \( k \) the number of trees, \( p \) the number of distinct clusters, and \( q \) the number of clusters (including repetitions).

• M-PHYLIP: The majority rule consensus tree method in PHYLIP [10]. It counts the occurrences of each cluster using hashing, and constructs the consensus tree from the clusters that occur more
than $\frac{k}{2}$ times. Since hashing is used, this method has expected time complexity $O(nk)$.

- **M-SumTrees**: The majority rule consensus tree method in SumTrees, which is part of DendroPy [24]. The documentation for the implemented algorithm was unavailable.

- **M-Naïve**: A self-implemented, naïve algorithm for computing the majority rule consensus tree, based on [27]. Given $S = \{T_1, \ldots, T_k\}$, it runs Day’s algorithm (see Section 2.1) $O(k)$ times, using each tree in $S$ as the reference tree $T_{ref}$ and comparing it against all others to count the occurrences of clusters. A consensus tree is constructed from those clusters that appear more than $\frac{k}{2}$ times. The time complexity of this algorithm is $O(nk^2)$.

- **M-Fast**: An implementation of our new majority rule consensus tree algorithm described in Sections 3 and 6.1. Its time complexity is $O(nk \log k)$.

- **L-Naïve**: A self-implemented, naïve algorithm for computing the loose consensus tree. First, all clusters in the input trees are extracted as bit vectors and the distinct clusters are retrieved. Every pair of distinct clusters is checked for pairwise compatibility, and the set of clusters compatible with all other clusters is then used to construct the consensus tree. Applying the constant optimizations mentioned in Section 6.1 gives a time complexity $O(\frac{nq}{\ell} + \frac{\ell^2 n}{\ell} + n^2)$. For this implementation, we set $\ell = 60$.

- **L-Fast**: An implementation of our new loose consensus tree algorithm described in Sections 4 and 6.1. Its time complexity is $O(nk)$.

- **G-PHYLIP**: The greedy consensus tree method in PHYLIP [10]. Like M-PHYLIP, the occurrences of the clusters are counted by hashing. Then, the clusters are processed in non-increasing order of the number of occurrences and a maximal set of pairwise compatible clusters is created. Checking whether two clusters are compatible is sped up to $O(\frac{nq}{\ell})$ by using words of length $\ell$. The expected time complexity is $O(q + \frac{n^2 q}{\ell} + n^2)$. The worst-case running time becomes $O(\frac{\ell^2}{\ell} nq + \frac{n^2}{\ell} + n^2) = 60$.

- **G-Fast**: A self-implemented, naive algorithm for computing the majority rule consensus tree algorithm described in Sections 5 and 6.1. Its time complexity is $O(\frac{nq}{\ell} + \frac{\ell^2 n}{\ell} + n^2)$. For this implementation, we set $\ell = 60$.

In addition to the above, the program COMPO-MENT [20] was also considered. This software uses hashing to compute its results. However, COMPO-MENT seems to have a built-in limit on the number of leaves, and crashes when $n > 100$. For this reason, it was not evaluated in our experiments.

**Testing**: All experiments were carried out on Ubuntu Nutty Narwhal, a 64-bit operating system with 8.00 GB RAM, and a CPU running at 2.20 GHz.

We used the following combinations of the parameters $n$ and $k$:

- (a) $n = 500$, $k = 1000$
- (b) $n = 1000$, $k = 500$
- (c) $n = 2000$, $k = 1000$
- (d) $n = 5000$, $k = 100$
- (e) $n \in \{500, 1000, 2000, 3000, 4000, 5000\}$, $k = 100$

For each of (a)–(d), we generated 10 data sets, applied the methods, and measured their running times. The worst-case and average running times (in seconds) are reported below. The purpose of case (d) was to demonstrate that our method M-Fast is much faster than M-PHYLIP when $n \gg k$. (The reason is that in this case, the log $k$-factor in the running time becomes almost negligible.) Thus, we did not run the other methods for (d). In (e), we generated at least 3 data sets for each specified value of $n$ and plotted the methods’ worst-case running times against each other in order to visualize the differences between them for a small, fixed value of $k$.

**Experimental results:**

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

<table>
<thead>
<tr>
<th></th>
<th>Worst-case</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-PHYLIP</td>
<td>1.94</td>
<td>1.88</td>
</tr>
<tr>
<td>M-SumTrees</td>
<td>91.18</td>
<td>89.55</td>
</tr>
<tr>
<td>M-Naïve</td>
<td>291.19</td>
<td>274.96</td>
</tr>
<tr>
<td>M-Fast</td>
<td>8.10</td>
<td>8.00</td>
</tr>
<tr>
<td>L-Naïve</td>
<td>8.00</td>
<td>7.12</td>
</tr>
<tr>
<td>L-Fast</td>
<td>5.34</td>
<td>5.16</td>
</tr>
<tr>
<td>G-PHYLIP</td>
<td>2.94</td>
<td>2.67</td>
</tr>
<tr>
<td>G-Naïve</td>
<td>4.34</td>
<td>4.14</td>
</tr>
<tr>
<td>G-Fast</td>
<td>4.10</td>
<td>3.76</td>
</tr>
</tbody>
</table>
(b) $n = 1000, k = 500$:

<table>
<thead>
<tr>
<th></th>
<th>Worst-case</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-PHYLIP</td>
<td>3.50</td>
<td>3.19</td>
</tr>
<tr>
<td>M-SumTrees</td>
<td>134.62</td>
<td>131.77</td>
</tr>
<tr>
<td>M-Na&quot;ive</td>
<td>138.23</td>
<td>134.69</td>
</tr>
<tr>
<td>M-Fast</td>
<td>7.54</td>
<td>7.38</td>
</tr>
<tr>
<td>L-Na&quot;ive</td>
<td>26.88</td>
<td>24.80</td>
</tr>
<tr>
<td>L-Fast</td>
<td>5.33</td>
<td>5.15</td>
</tr>
<tr>
<td>G-PHYLIP</td>
<td>6.56</td>
<td>5.99</td>
</tr>
<tr>
<td>G-Na&quot;ive</td>
<td>11.55</td>
<td>10.75</td>
</tr>
<tr>
<td>G-Fast</td>
<td>6.59</td>
<td>6.23</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Worst-case</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-PHYLIP</td>
<td>34.07</td>
<td>30.03</td>
</tr>
<tr>
<td>M-SumTrees</td>
<td>932.12</td>
<td>918.55</td>
</tr>
<tr>
<td>M-Na&quot;ive</td>
<td>1100.69</td>
<td>1089.57</td>
</tr>
<tr>
<td>M-Fast</td>
<td>32.24</td>
<td>31.96</td>
</tr>
<tr>
<td>L-Na&quot;ive</td>
<td>335.31</td>
<td>319.03</td>
</tr>
<tr>
<td>L-Fast</td>
<td>22.11</td>
<td>21.85</td>
</tr>
<tr>
<td>G-PHYLIP</td>
<td>67.19</td>
<td>63.09</td>
</tr>
<tr>
<td>G-Na&quot;ive</td>
<td>115.72</td>
<td>111.78</td>
</tr>
<tr>
<td>G-Fast</td>
<td>41.32</td>
<td>40.08</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Worst-case</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-PHYLIP</td>
<td>93.25</td>
<td>90.04</td>
</tr>
<tr>
<td>M-SumTrees</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>M-Na&quot;ive</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>M-Fast</td>
<td>6.41</td>
<td>6.27</td>
</tr>
<tr>
<td>L-Na&quot;ive</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>L-Fast</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>G-PHYLIP</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>G-Na&quot;ive</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>G-Fast</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

(e) $n \in \{500, 1000, 2000, 3000, 4000, 5000\}$, $k = 100$:

In the following three diagrams, the horizontal axis represents $n$ and the vertical axis represents the worst-case running time (in seconds).

**Discussion:** Based on the experimental results, we see that the improved consensus tree algorithms perform much better than their naive counterparts, as expected. We also see that our prototype implementations are competitive against the currently available software, even though our algorithms do not use any randomization.

- Our improved majority consensus tree algorithm performed better than SumTrees and COMPONENT for all data sets. Furthermore, it was significantly faster than PHYLIP when $n$ was large and
$k$ was small ($n = 5000$, $k = 100$). On the other hand, for small $n$, PHYLIP was better. For the case $n = 2000$, $k = 1000$, they had roughly the same running times, with PHYLIP being slightly faster on average and our algorithm being slightly faster in the worst case.

- Our improved loose consensus tree algorithm could handle much larger data sets than COMPONENT and ran quickly, producing a solution for the data set with $n = 2000$, $k = 1000$ in a little over 20 seconds.

- Our improved greedy consensus tree algorithm was slower than PHYLIP when $n$ and $k$ were small and $n \ll k$. However, it outperformed PHYLIP as the data sets got larger and $n \gg k$.

We conclude that hashing is not always necessary to obtain fast algorithms for constructing consensus trees.

7 Final remarks
To end this paper, we briefly mention a few other useful types of consensus trees and some related open problems. As above, let $S = \{T_1, T_2, \ldots, T_k\}$ be a set of trees satisfying $\Lambda(T_1) = \Lambda(T_2) = \cdots = \Lambda(T_k) = L$ for some leaf label set $L$ of cardinality $n$.

First, a strict consensus tree of $S$ [23] is a tree $T$ with $\Lambda(T) = L$ containing precisely those clusters that occur in every tree in $S$, i.e., $\mathcal{C}(T) = \bigcap_{i=1}^{k} \mathcal{C}(T_i)$. This type of consensus tree is well understood [5, 9, 25]. The advantages of the strict consensus tree is that it is always unique and can be computed quickly; the algorithm by Day [7] (see Section 2.1) can compute it in (optimal) $O(nk)$ time. The disadvantage of the strict consensus tree is that it often discards valuable branching information. For example, in Figure 1, only the trivial clusters occur in every tree in $S$, so the strict consensus tree of $S$ is just a root node directly attached to the leaves $a, b, c, d, e$.

Secondly, an $R^*$ consensus tree of $S$ [5] is a tree $T$ with $\Lambda(T) = L$ that contains as embedded subtrees as many so-called rooted triplets as possible from a special set $R_{maj}$ and no other rooted triplets; see [5, 8, 13] for the definition. An $R^*$ consensus tree provides a statistically consistent estimator of the species tree topology when combining a set of gene trees [8]. On the negative side, it is still not known how to compute it efficiently. The currently fastest methods run in $O(n^k)$ time for unbounded $k$ [5, 13] and in $O(n^2 \sqrt{\log n})$ time when $k = 2$ [13]. It is an open problem to reduce their running times.

Thirdly, extensions of consensus trees to multi-labeled phylogenetic trees (MUL-trees), where the same leaf label may be used more than once in the same tree, were recently introduced by [15] and further studied in [6, 12]. Here, a major obstacle is that MUL-trees’ cluster collections are no longer sets but multisets, and certain basic problems become NP-hard when extended to multisets. An important task is to define informative types of consensus MUL-trees that admit efficient algorithms.

For further discussions on the advantages and disadvantages of different types of consensus trees, see [5, 8, 9, 25].

Acknowledgments
The authors would like to thank Joseph Felsenstein for clarifications regarding PHYLIP and Todd Wareham for confirming the time complexity of the deterministic algorithm in [27].

References


