Improved Algorithms for Constructing Consensus Trees

JESPER JANSSON, Kyoto University
CHUANQI SHEN, Stanford University
WING-KIN SUNG, National University of Singapore

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 article 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 $O(nk)$ time (majority rule consensus tree), $O(nk)$ time (loose consensus tree), and $O(n^2k)$ time (greedy consensus tree). Our algorithms for the majority rule consensus and the loose consensus trees are optimal since the input size is $\Omega(nk)$. Experimental results show that the algorithms are fast in practice.

Categories and Subject Descriptors: F.2.2 [**Nonnumerical Algorithms and Problems**]: Computations on Discrete Structures; G.2.2 [**Graph Theory**]: Trees

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Phylogenetic tree, cluster, pairwise compatibility, majority rule consensus tree, loose consensus tree, greedy consensus tree, implementation

ACM Reference Format:
Jesper Jansson, Chuanqi Shen, and Wing-Kin Sung. 2016. Improved algorithms for constructing consensus trees. J. ACM 63, 3, Article 28 (June 2016), 24 pages.
DOI: http://dx.doi.org/10.1145/2925985

1. INTRODUCTION

Scientists and scholars often use *phylogenetic trees* to describe evolutionary relationships [Felsenstein 2004; Gusfield 1997; Nakhleh et al. 2005; Semple and Steel 2003; Sung 2010]. Since the early 1860s, a vast number of phylogenetic trees have been constructed and published in the literature but they do not always agree with each other; two trees based on different datasets or obtained by different methods may contain contradicting branching patterns even though their leaf label sets are identical. Also,

The results in this article appeared in preliminary form in *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA 2013), pp. 1800–1813, SIAM, 2013, and in *Proceedings of the 17th Annual International Conference on Research in Computational Molecular Biology* (RECOMB 2013), Lecture Notes in Computer Science, Vol. 7821, pp. 88–99, Springer-Verlag, 2013. J. J. was funded by The Hakubi Project at Kyoto University and KAKENHI Grant No. 26330014.
Authors' addresses: J. Jansson, Laboratory of Mathematical Bioinformatics (Akutsu Laboratory), Institute for Chemical Research, Kyoto University, Gokasho, Uji, Kyoto 611-0011, Japan; email: jj@kuicr.kyoto-u.ac.jp; C. Shen, Stanford University, 450 Serra Mall, Stanford, CA 94305-2004; email: shencq@stanford.edu; W.-K. Sung, 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; email: ksung@comp.nus.edu.sg.

ACM 0004-5411/2016/06-ART28 $15.00
DOI: http://dx.doi.org/10.1145/2925985

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 [Amenta et al. 2003; Bansal et al. 2011; Degnan et al. 2009; Felsenstein 2004; Kannan et al. 1998; Sung 2010]. To deal with conflicts that arise between two or more such trees in a systematic manner, the concept of a *consensus tree* was invented [Adams III 1972; Bryant 2003]. Informally, a consensus tree is a phylogenetic tree that 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 [Felsenstein 2004] or as a basis for similarity measures between two given phylogenetic trees.[1]

There are many ways to reconcile structural differences and remove inconsistencies in a collection of trees. Depending on the application and the quality of the input data, different definitions of a "consensus tree" may be appropriate. Consequently, several alternatives have been proposed and analyzed by biologists, mathematicians, and computer scientists since the 1970s; see, for example, Bryant [2003], Chapter 30 in Felsenstein [2004], or Chapter 8.4 in Sung [2010] for some surveys. Three of the most widely used ones among practitioners are the following:

(i) *the majority rule consensus tree* [Margush and McMorris 1981],
(ii) *the loose consensus tree* [Meacham 1982 (see p. 84 of Jensen [1983]) and [Bremer 1990]], and
(iii) *the greedy consensus tree* [Bryant 2003; Felsenstein 2005].

For example, a search on Google Scholar for "majority rule consensus tree" returns thousands of articles published in biology-related journals using this concept. Indeed, in some contexts, the majority rule consensus tree can be regarded as an optimal summary of a collection of trees [Holder et al. 2008]. See also the introduction of Cotton and Wilkinson [2007] for other uses of the majority rule consensus tree.

Popular computational phylogenetics software packages such as PHYLIP [Felsenstein 2005] and MrBayes [Ronquist and Huelsenbeck 2003] contain implementations for constructing (i) and (iii); COMPONENT [Page 1993] implements (i) and (ii); SumTrees in DendroPy [Sukumaran and Holder 2010] implements (i); and PAUP* [Swofford 2003] implements (i), (ii), and (iii). Although these programs work very well in practical applications, 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. Therefore, in this article, we develop new, simple deterministic algorithms for constructing (i), (ii), and (iii). Our new algorithms are fast, both in theory and in practice.

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 onward, and every leaf in a tree is identified with its (unique) label. All edges in a tree are directed from the root of the tree to its leaves. 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

[1]Measuring the similarity between phylogenetic trees is useful, for example, when querying phylogenetic databases [Bansal et al. 2011] or evaluating methods for phylogenetic reconstruction [Kuhner and Felsenstein 1994].

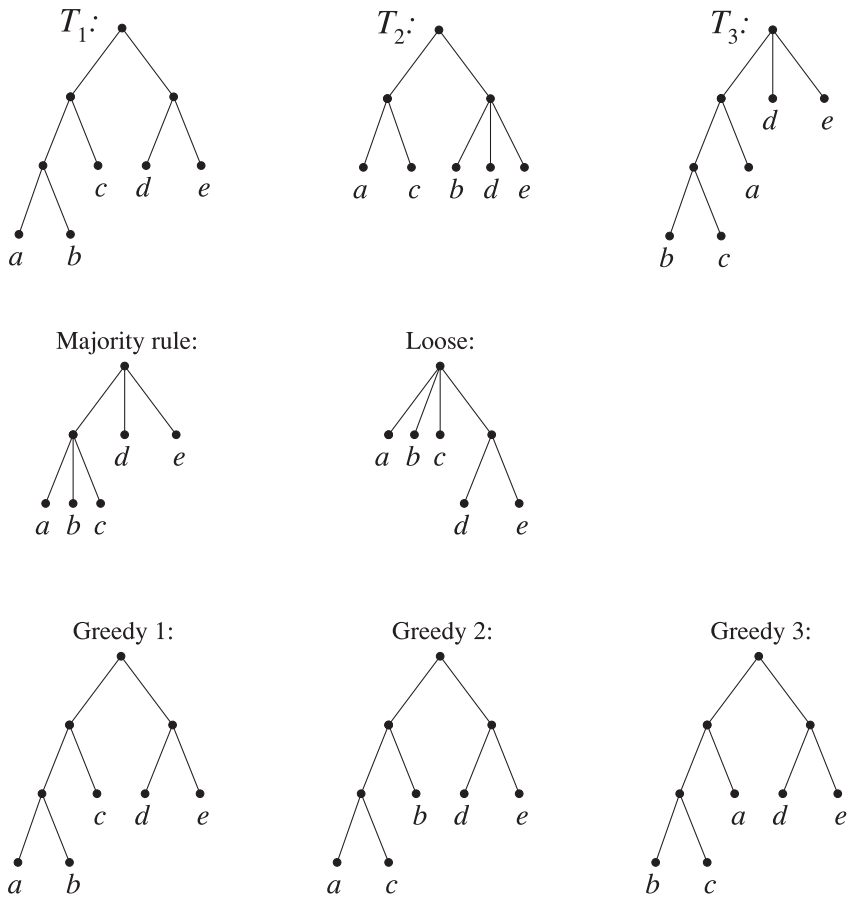


Fig. 1. Let $\mathcal{S} = \{T_1, T_2, T_3\}$ as shown above with $L = \Lambda(T_1) = \Lambda(T_2) = \Lambda(T_3) = \{a, b, c, d, e\}$. The cluster collections of T_1, T_2, and T_3 are as follows:
$\mathcal{C}(T_1) = \{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{a, b\}, \{a, b, c\}, \{d, e\}, L\}$,
$\mathcal{C}(T_2) = \{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{a, c\}, \{b, d, e\}, L\}$,
$\mathcal{C}(T_3) = \{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{b, c\}, \{a, b, c\}, L\}$,
Majority rule, loose, and greedy consensus trees of $\mathcal{S}$ are displayed. Observe that the only non-trivial majority cluster in $\mathcal{S}$ is $\{a, b, c\}$. Also observe that $\{d, e\}$ is the only non-trivial cluster in $\mathcal{S}$ that is compatible with all trees in $\mathcal{S}$.

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 *lowest common ancestor of S in T*, denoted by $lca^T(S)$, 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 non-empty subset 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, and if u is a leaf, then $\Lambda(T[u])$ is a singleton set. For any $C \subseteq \Lambda(T)$, if $|C| = 1$ or $C = \Lambda(T)$, then C is called *trivial*; otherwise, C is *non-trivial*. The *cluster collection of T* is defined as $\mathcal{C}(T) = \bigcup_{u \in V(T)} \{\Lambda(T[u])\}$. See Figure 1 for some examples. When a cluster $C \subseteq \Lambda(T)$ belongs to $\mathcal{C}(T)$, 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 $u \in V(T)$. (As an example, in Figure 1, the cluster $\{b, d\}$ is compatible with T_2, but not compatible with T_1 and T_3.) 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*.

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 *consensus tree for* $\mathcal{S}$ is a tree that summarizes the branching information contained in $\mathcal{S}$ according to some well-defined rule. This article focuses on the following three variants:

- A cluster that occurs in more than $k/2$ of the trees in $\mathcal{S}$ is a *majority cluster of* $\mathcal{S}$. A *majority rule consensus tree of* $\mathcal{S}$ is a tree T such that $\Lambda(T) = L$ and $\mathcal{C}(T)$ consists of all majority clusters of $\mathcal{S}$.
- A *loose consensus tree of* $\mathcal{S}$ is a tree T such that $\Lambda(T) = L$ and $\mathcal{C}(T)$ consists of all clusters that occur in at least one tree in $\mathcal{S}$ and that are compatible with all trees in $\mathcal{S}$.
- Let $\mathcal{X}$ be a list of all clusters that occur in at least one tree in $\mathcal{S}$, sorted according to the number of trees in $\mathcal{S}$ in which they occur (frequently occurring clusters coming first and with ties broken arbitrarily). Construct a set $\mathcal{Y}$ of clusters as follows: Initialize $\mathcal{Y} := \emptyset$. Then, traverse the list $\mathcal{X}$ and for each cluster C encountered in this order and check if C and C' are pairwise compatible for all $C' \in \mathcal{Y}$; if yes, then let $\mathcal{Y} := \mathcal{Y} \cup \{C\}$. A *greedy consensus tree of* $\mathcal{S}$ is a tree T such that $\Lambda(T) = L$ and $\mathcal{C}(T) = \mathcal{Y}$.

An example is given in Figure 1. As pointed out in Bryant [2003], for any given $\mathcal{S}$, there exists a unique majority rule consensus tree of $\mathcal{S}$ and a unique loose consensus tree of $\mathcal{S}$, but a greedy consensus tree of $\mathcal{S}$ is not always uniquely defined. (In the example in Figure 1, three different greedy consensus trees exist because each of the clusters $\{a, b\}$, $\{a, c\}$, $\{b, c\}$ occurs once in $\mathcal{S}$ and exactly one of them will be included in any greedy consensus tree, depending on how ties among clusters are broken.) Moreover, if a cluster C occurs in the majority rule consensus tree of $\mathcal{S}$ or in the loose consensus tree of $\mathcal{S}$, then C occurs in every greedy consensus tree of $\mathcal{S}$.

Throughout the text, we shall use the following notation to express the time complexities of algorithms for constructing consensus trees. Let $\mathcal{S}$ be an input set of trees with identical leaf label sets. Define $k = |\mathcal{S}|$ and $n = |L|$, and write $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$, 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 $\mathcal{S}$ and q the total number of clusters occurring in $\mathcal{S}$ (with repetitions). Thus, $p \leq q$ and $q = \Theta(nk)$ with $k \cdot (n+1) \leq q \leq k \cdot (2n-1)$.

1.2. Previous Work

The majority rule consensus tree was introduced by Margush and McMorris [1981]. Wareham [1985] gave a deterministic algorithm with a worst-case running time of $O(n^2 + nk^2)$ for building it.

The loose consensus tree is also known as *the combinable component consensus tree* and *the semi-strict consensus tree*. It was introduced by Meacham in 1982 (see p. 84 of Jensen [1983]) and independently by Bremer [1990] and can be computed in $O(nq^2) = O(n^3k^2)$ time by a method outlined by McMorris and Wilkinson [2011] that tests every cluster occurring in $\mathcal{S}$ against all other clusters in $\mathcal{S}$ for compatibility. In the special case where T_i and T_j are compatible for all $T_i, T_j \in \mathcal{S}$, the main algorithm in Warnow [1994] constructs the loose consensus tree of $\mathcal{S}$ in $O(nk)$ time.

The greedy consensus tree is sometimes called *the majority rule extended consensus tree* in the literature because it can be computed by allowing additional (non-majority) clusters to be inserted into the majority rule consensus tree in a greedy fashion [Bryant 2003; Felsenstein 2005]. The straightforward algorithm implied by the definition of a greedy consensus tree in Section 1.1 above (originally from Section 2.1.4 in Bryant's survey [2003]) runs in $O(nq + n^2p) = O(n^3k)$ time.

As for *randomized* methods, Amenta et al. [2003] published an algorithm for the majority rule consensus tree with $O(nk)$ expected running time 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 consensus trees in existing software packages such as PHYLIP [Felsenstein 2005], MrBayes [Ronquist and Huelsenbeck 2003], SumTrees in DendroPy [Sukumaran and Holder 2010], COMPONENT [Page 1993], and PAUP* [Swofford 2003] also rely on randomization and typically have unbounded worst-case running times as well.

Some previous results related to other types of consensus trees are discussed in Section 7.

1.3. New Results

After describing a number of essential algorithmic tools and properties of trees in Section 2, 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 $\mathcal{S}$ of trees with identical leaf label sets. The worst-case running times of the previously fastest deterministic algorithms (not including the ones in the preliminary version of this article[2]) and our new ones are compared below:

.

	Previously best	This article
Majority rule consensus tree	$O(n^2 + nk^2)$ time [Wareham 1985]	$O(nk)$ time Section 3
Loose consensus tree	$O(nq^2) = O(n^3k^2)$ time [McMorris and Wilkinson 2011]	$O(nk)$ time Section 4
Greedy consensus tree	$O(nq + n^2p) = O(n^3k)$ time [Bryant 2003]	$O(nq) = O(n^2k)$ time Section 5

Our algorithms for the majority rule consensus tree and the loose consensus tree are optimal since the size of the input is $\Omega(nk)$. We thus resolve two long-standing open problems in phylogenetics.

We implemented our algorithms to make sure that they are practical and applied them to various simulated datasets, as explained in Section 6. In short, these experiments showed 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 [Felsenstein 2005], without having to use randomization and hash tables for storing the clusters occurring in $\mathcal{S}$. Notably, for inputs consisting of a small number of very large trees (i.e., $n \gg k$), our prototype implementations were much faster than than PHYLIP. In contrast to current practice,

[2]In the preliminary version of this article [Jansson et al. 2013], we developed a deterministic algorithm for the majority rule consensus tree with $O(nk \log k)$ worst-case running time, based on recursion.

this suggests that it might not always be a good idea to use randomization and hashing when computing consensus trees.

2. PRELIMINARIES

This section lists some algorithmic results and properties of trees that will be used later.

2.1. Day's Algorithm

Day's algorithm [Day 1985] takes as input two trees T_{ref} and T with identical leaf label sets. 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. In particular, Day's algorithm can be applied to identify the set of all clusters that occur in both T_{ref} and T in $O(n)$ time, where $n = |L|$.

It works as follows. In the preprocessing phase, it does an $O(n)$-time depth-first traversal of T_{ref} while enumerating all the leaves as they are encountered. This yields a bijection f from L to the set $\{1, 2, \ldots, n\}$ under which every $C \in \mathcal{C}(T_{ref})$ forms an interval of consecutive integers. Furthermore, each of the at most $n-1$ intervals that represents a non-singleton cluster in $\mathcal{C}(T_{ref})$ (or, equivalently, each internal node of T_{ref}) is assigned to one of the n leaves in T_{ref} in such a way that (1) no leaf gets more than one interval and (2) any interval $[a..b]$ is assigned to either the leaf $f^{-1}(a)$ or the leaf $f^{-1}(b)$. One way to do so is by applying a rule of the following form: For each internal node u in T_{ref}, if u has no left sibling, then assign u to the rightmost leaf descendant of u; otherwise, assign u to the leftmost leaf descendant of u. Next, the algorithm preprocesses T in $O(n)$ time to store $f(x)$ in each leaf x of T and to do a bottom-up traversal of T to obtain, for every $u \in V(T)$, the values $m(u) := \min_{x \in \Lambda(T[u])}\{f(x)\}$, $M(u) := \max_{x \in \Lambda(T[u])}\{f(x)\}$, and $size(u) := |\Lambda(T[u])|$.

After the preprocessing is done, one can check for any specified internal node u in T whether or not the cluster $\Lambda(T[u])$ occurs in T_{ref} using $O(1)$ time simply by checking (1) if $size(u) = M(u) - m(u) + 1$ (i.e., if the interval $[m(u)..M(u)]$ is an interval of consecutive integers) and (2) if either one of the two leaves $f^{-1}(m(u))$ and $f^{-1}(M(u))$ in T_{ref} was assigned the interval $[m(u)..M(u)]$. We summarize this result as follows:

THEOREM 2.1 ([DAY 1985]). *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. The *delete* and *insert* Operations on a Tree

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 $\mathcal{C}(T)$ without affecting the other clusters. The time needed for this operation is proportional to the number of children of u.

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 $\mathcal{C}(T)$.

2.3. Characterizing 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.

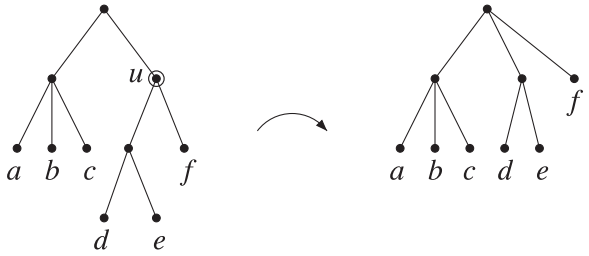


Fig. 2. Let T be the tree on the left and let u be the marked node. Then $\Lambda(T[u]) = \{d, e, f\}$ and applying the *delete* operation on u removes the cluster $\{d, e, f\}$ from $\mathcal{C}(T)$. The remaining non-trivial clusters are $\{a, b, c\}$ and $\{d, e\}$.

LEMMA 2.2. *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 $Child(C)$, i.e., $z \notin \Lambda(T[c_i])$. But then $\{x, z\} \subseteq C$ and $y \notin C$, while $\{x, y\} \subseteq \Lambda(T[c_i])$ and $z \notin \Lambda(T[c_i])$, so C and $\Lambda(T[c_i])$ are not pairwise compatible. By definition, C is not compatible with T.

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

- u is an ancestor of $lca^T(C)$: Then trivially $C \subseteq \Lambda(T[u])$.
- u is a descendant of $lca^T(C)$: Let c_i be the child of $lca^T(C)$ 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 $lca^T(C)$: Then $C \cap \Lambda(T[u]) = \emptyset$.

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

2.4. Procedure `Merge_Trees`

`Merge_Trees` is a procedure that combines all the clusters from two non-conflicting trees into one tree in linear time. Formally, let T_1 and T_2 be two trees with $\Lambda(T_1) = \Lambda(T_2) = L$ such that T_1 and T_2 are compatible. 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, where $n = |L|$.[3]

`Merge_Trees` operates in two phases. The first phase is a preprocessing phase that works as follows. As in Day's algorithm (see Section 2.1), do an $O(n)$-time depth-first traversal of T_1 to construct a bijection f from L to the set $\{1, 2, \ldots, n\}$ under which every $C \in \mathcal{C}(T_1)$ forms an interval of consecutive integers. Do a bottom-up traversal of T_2 to obtain and store, for each $v \in V(T_2)$, the value $m(v) := \min_{x \in \Lambda(T_2[v])}\{f(x)\}$. Also do a 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 $depth(v)$. Then, transform T_2 into an ordered tree by ordering the children at each internal node v of T_2 so for every two children a and b of v, a is to the left of b if and only if $m(a) < m(b)$. This can be done in $O(n)$ time in total by putting all $v \in V(T_2)$ into a single list $\mathcal{X}$, sorting $\mathcal{X}$ using the $m(v)$-values as the key (to sort $O(n)$ integers belonging to $\{1, 2, \ldots, n\}$ takes $O(n)$ time with counting sort), and then traversing the sorted $\mathcal{X}$ to set the left-to-right orderings of the children at all nodes in T_2. Now Lemma 2.2 implies:

[3]The procedure INSERT in Warnow [1994] also accomplishes this, but in our opinion, `Merge_Trees` is simpler.

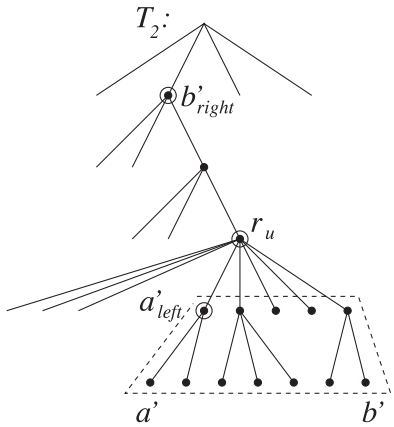


Fig. 3. Suppose that $\Lambda(T_1[u])$ for some specified $u \in V(T_1)$ corresponds to the interval $[a..b]$ in the left-to-right leaf ordering in T_2. The relationship between the nodes a'_{left}, b'_{right}, and r_u determines where in T_2 to insert $\Lambda(T_1[u])$ as a new cluster. In this example, $d_u = a'_{left}$ and $e_u =$ the rightmost child of r_u.

LEMMA 2.3. *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 $C = \bigcup_{c_i \in D} \Lambda(T_2[c_i])$, where D is a consecutive subsequence of the children of the node $lca^{T_2}(C)$.*

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 $r_u := lca^{T_2}(\Lambda(T_1[u])$ 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 $leaf\_rank_{T_2}(x)$ to be $1+$ (the number of leaves to the left of x in T_2). Then, for every $u \in V(T_1)$, define $start(u) := \min_{x \in \Lambda(T_1[u])} leaf\_rank_{T_2}(x)$ and $stop(u) := \max_{x \in \Lambda(T_1[u])} leaf\_rank_{T_2}(x)$. Intuitively, $start(u)$ and $stop(u)$ 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 formulas to precompute $start(u)$ and $stop(u)$ for all $u \in V(T_1)$ in $O(n)$ time in total:

LEMMA 2.4. *For any $u \in V(T_1)$, let $Child(u)$ be the set of children of u. Then:*

$$start(u) = \begin{cases} leaf\_rank_{T_2}(u), & \text{if } u \text{ is a leaf} \\ \min_{c \in Child(u)} start(c), & \text{otherwise} \end{cases}$$

$$stop(u) = \begin{cases} leaf\_rank_{T_2}(u), & \text{if } u \text{ is a leaf} \\ \max_{c \in Child(u)} stop(c), & \text{otherwise} \end{cases}$$

Next, for every $x \in L$, define x_{left} as the node v in T_2 with the smallest value of $depth(v)$ (i.e., as close to the root as possible) whose leftmost leaf descendant is x. Define x_{right} for every $x \in L$ analogously but using the rightmost leaf descendant instead. See Figure 3. To compute x_{left} and x_{right} for all $x \in L$, do an $O(n)$-time bottom-up traversal of T_2. Finally, apply the method of Bender and Farach-Colton [2000] or Harel and Tarjan [1984] to preprocess T_2 in $O(n)$ time so 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 `Merge_Trees` which inserts clusters from T_1 into T_2. (Recall from the first paragraph in this subsection that `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 $a := start(u)$ and $b := stop(u)$, and let a' and b' be the elements of L satisfying $leaf\_rank_{T_2}(a') = a$ and $leaf\_rank_{T_2}(b') = b$. Obtain $r_u := lca^{T_2}(\{a', b'\})$ in $O(1)$ time by querying the *lca* data structure. Define d_u as the leftmost child of r_u such that $\Lambda(T_1[u]) \cap \Lambda(T_2[d_u]) \neq \emptyset$ and e_u as the rightmost child of r_u such that $\Lambda(T_1[u]) \cap \Lambda(T_2[e_u]) \neq \emptyset$. (As will be shown shortly, d_u and e_u tell us where in T_2 to insert $\Lambda(T_1[u])$ as a new cluster.) See Figure 3 for an illustration. To compute d_u and e_u efficiently, we can use the next lemma.

LEMMA 2.5. *The following holds:*

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

Thus, apply Lemma 2.5 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 $\Lambda(T_2[r_u]) = \Lambda(T_1[u])$, that is, the cluster already occurs in T_2 and we do nothing. Otherwise, insert $\Lambda(T_1[u])$ into T_2 by creating a new child c of r_u, setting $depth(c) := depth(r_u) + 1$, and letting all children of r_u in the sequence $d_u, \ldots, e_u$ become children of c. Also update a'_{left} to point to c if d_u was not the leftmost child of r_u, and update b'_{right} analogously. The correctness follows from Lemma 2.3. Finally, after all nodes have been taken care of, return T_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.

THEOREM 2.6. *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.*

2.5. Procedure `One-Way_Compatible`

This subsection describes a linear-time procedure named `One-Way_Compatible` whose input is two trees T_1 and T_2 with identical leaf label sets and whose output is a copy of T_1 in which every cluster that is not compatible with T_2 has been removed. In other words, for any two trees T_1 and T_2 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 $\mathcal{C}(T) = \{C \in \mathcal{C}(T_1) :$ C is compatible with $T_2\}$. 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`$(T_2, T_1) = T_2$.

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 from 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 $a := start(u)$ and $b := stop(u)$, and let a' and b' be the elements of L such that $leaf\_rank_{T_2}(a') = a$ and $leaf\_rank_{T_2}(b') = b$. Compute $r_u := lca^{T_2}(\{a', b'\})$ in $O(1)$ time by querying the *lca* data structure. Next, if $depth(a'_{left}) > depth(r_u)$, then define $d_u := a'_{left}$; otherwise, define $d_u :=$ the leftmost child of r_u. Similarly,

if $depth(b'_{right}) > depth(r_u)$, then define $e_u := b'_{right}$; 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 2.7. *$\Lambda(T_1[u])$ 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 2.3 states that C is compatible with T_2 if and only if $C = \Lambda(T_2[c_i]) \cup \Lambda(T_2[c_{i+1}]) \cup \ldots \cup \Lambda(T_2[c_j])$ for some consecutive subsequence $c_i, c_{i+1}, \ldots, c_j$ of the children of the node r_u.

($\rightarrow$) Suppose $C = \Lambda(T_2[c_i]) \cup \Lambda(T_2[c_{i+1}]) \cup \ldots \cup \Lambda(T_2[c_j])$, where $c_i, c_{i+1}, \ldots, c_j$ 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 $d_u = a'_{left}$ 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'_{left}, the leftmost leaf descendant of every node on the path in T_2 between a' and a'_{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 $C \subseteq \Lambda(T_2[d_u]) \cup \ldots \cup \Lambda(T_2[e_u])$, 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 $|\Lambda(T_2[d_u]) \cup \ldots \cup \Lambda(T_2[e_u])| = b - a + 1$. From condition (iii), $|C| = |\Lambda(T_2[d_u]) \cup \ldots \cup \Lambda(T_2[e_u])|$, which shows that $C = \Lambda(T_2[d_u]) \cup \ldots \cup \Lambda(T_2[e_u])$, where $d_u, \ldots, e_u$ is a consecutive subsequence of children of r_u. □

Now, the second phase of `One-Way_Compatible` is as follows: For each $u \in V(T_1)$, apply Lemma 2.7 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 $O(n)$ since each compatibility check takes $O(1)$ time by applying Lemma 2.7 and since the total time needed for all node deletions is $O(n)$. The latter is because whenever a node u in T_1 is deleted so the children of u get a new parent, the top-down order ensures that the new parent will never be deleted; hence, for every node in T_1, its parent can change at most once.

THEOREM 2.8. *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 $\mathcal{C}(T) = \{C \in \mathcal{C}(T_1) : \text{C is compatible with } T_2\}$ in $O(n)$ time.*

2.6. Finding All Majority Elements

In this subsection, 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 $\mathcal{W}$, that is, for each $j \in \{1, 2, \ldots, k\}$, consider $\mathcal{W}[j]$ and do the following. First, 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. Second, insert every $x \in \mathcal{W}[j]$ that 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 $\mathcal{W} = (\mathcal{W}[1], \mathcal{W}[2], \mathcal{W}[3]) = (\{a, b, d\}, \{a, c\}, \{d, e\})$. After the first iteration of Phase 1, the set of current candidates is $\{a, b, d\}$ and all three counters are set to 1. After the second iteration, the set of current candidates is $\{a, c\}$, with a's and c's counters equal to 2 and 1, respectively. After the last iteration of Phase 1, the set of current candidates is $\{a, d, e\}$. 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 2.9. *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 2.10. The algorithm presented above can be viewed as a direct extension of Boyer and Moore's algorithm in Boyer and Moore [1991], which solves the special case of the problem where every subset in the list $\mathcal{W}$ has cardinality 1.

3. CONSTRUCTING THE MAJORITY RULE CONSENSUS TREE

Here, we present a new algorithm `Maj_Rule_Cons_Tree` for building the majority rule consensus tree of $\mathcal{S}$. It uses the technique from Section 2.6 to locate all majority clusters in $\mathcal{S}$ by interpreting X as the set of all possible clusters of L (so 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 `Maj_Rule_Cons_Tree` has the same high-level structure as the two-phase algorithm in Section 2.6: In Phase 1, it computes a set of candidate clusters that includes all majority clusters, and then, in Phase 2, it removes all candidate clusters 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. Storing the candidate clusters in a tree instead of in a list is in fact the key to getting an efficient algorithm.

The pseudocode is summarized in Figure 4. Phase 1 and Phase 2 are described in Sections 3.1 and 3.2, respectively. To achieve a good time complexity, some steps of the

Algorithm `Maj_Rule_Cons_Tree`
Input: A set $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$ of trees with $\Lambda(T_1) = \Lambda(T_2) = \cdots = \Lambda(T_k)$.
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 $v \in V(T)$ in top-down order **do**
if $\Lambda(T[v])$ 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 $\Lambda(T[v]) = C$.
endfor
endfor

/* Phase 2 */
4 **for** each $v \in V(T)$ **do** $count(v) := 0$
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** $count(v) := count(v) + 1$
6 **for** each $v \in V(T)$ in top-down order **do**
if $count(v) \leq k/2$ **then** perform a *delete* operation on v.
7 **return** T
End `Maj_Rule_Cons_Tree`

Fig. 4. Algorithm `Maj_Rule_Cons_Tree` for constructing the majority rule consensus tree.

algorithm are implemented by applying Day's algorithm (i.e., to count occurrences of clusters) and the procedures `Merge_Trees` and `One-Way_Compatible` from Section 2 (i.e., to insert new candidate clusters into T that are compatible with the current T but not already in T); the details are given in Section 3.3.

3.1. 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 2.6, 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 `Maj_Rule_Cons_Tree` and the method in Section 2.6.

The first difference is that `Maj_Rule_Cons_Tree` does not store the set of current candidates in a sorted list as in Section 2.6 but encodes them as nodes in a tree T whose leaf label set equals L. 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 the 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 2.6 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 3.1. *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 the 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 treated, then C's counter will be decremented by 1.
- If C is not a current candidate when T_x is treated, 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. □

3.2. 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 3.1 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 3.2. *The tree output by Algorithm* `Maj_Rule_Cons_Tree` *at the end of Phase 2 is the majority rule consensus tree of $\mathcal{S}$.*

3.3. Time Complexity Analysis

We now analyze the worst-case running time of Algorithm `Maj_Rule_Cons_Tree`.

THEOREM 3.3. *Algorithm* `Maj_Rule_Cons_Tree` *constructs the majority rule consensus tree of $\mathcal{S}$ in $O(nk)$ time.*

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 (see Section 2.1) with $T_{ref} = T_j$ and then check each $\Lambda(T[v])$ to see if it occurs in T_j. By Theorem 2.1, this requires $O(n)$ time for preprocessing, and each of the $O(n)$ nodes in $V(T)$ can be checked in $O(1)$ time.[4] The *delete* operations take $O(n)$ time in total since every node's parent is changed at most once (the nodes are handled in top-down order, so if some node is deleted then the new parent of its children cannot be deleted 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), updating the structure of T to make T isomorphic to the obtained Q, and setting the counters of all new

[4]This way of counting occurrences of clusters has been used elsewhere in the literature, for example, in Wareham [1985] and on p. 217 of Sung [2010].

nodes to 1. This works because according to Theorem 2.8, P is a tree consisting of the clusters occurring in T_j that are compatible with the set of current candidates, and by Theorem 2.6, 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 by applying Day's algorithm like in Step 3.1. 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. □

Remark 3.4. A natural way to parameterize the majority rule consensus tree is by letting ℓ be any real number such that $1/2 \leq \ell \leq 1$ and keeping only clusters that occur in more than a fraction ℓ of the input trees in $\mathcal{S}$ [McMorris et al. 1983]. Algorithm `Maj_Rule_Cons_Tree` can be modified accordingly without affecting the time complexity by changing Step 6 since the set of all such clusters for any fixed $1/2 \leq \ell \leq 1$ is a subset of the set of majority clusters of $\mathcal{S}$.

4. CONSTRUCTING THE LOOSE CONSENSUS TREE

The loose consensus tree of $\mathcal{S}$ can be computed by testing every cluster that occurs in $\mathcal{S}$ against all other clusters in $\mathcal{S}$ for compatibility [McMorris and Wilkinson 2011]. Since each pair of clusters can be checked in $O(n)$ time, this gives an algorithm with $O(nq^2) = O(n^3k^2)$ running time. (If we incorporate a bottom-up technique based on Lemma 2.2 to check a cluster for compatibility with a tree in $O(n)$ time, then the running time is improved slightly to $O(nkq) = O(n^2k^2)$.) 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 and `One-Way_Compatible` from Section 2.5 as subroutines.

First, for any $j \in \{1, 2, \ldots, k\}$, we define the set of *one-way compatible clusters up to* j as the set $\mathcal{O}_j = \bigcup_{i=1}^{j} \{C \in \mathcal{C}(T_i) : C \text{ is compatible with all trees in } \{T_i, T_{i+1}, \ldots, T_j\}\}$. It is easy to see that:

LEMMA 4.1. *For any $j \in \{1, 2, \ldots, k\}$, all clusters in $\mathcal{O}_j$ are pairwise compatible.*

PROOF. Consider any two clusters $C, C' \in \mathcal{O}_j$. 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 $C \in \mathcal{C}(T_i)$ and $C' \in \mathcal{C}(T_{i'})$, where $i < i' \leq j$. Since $C \in \mathcal{O}_j$, 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 Semple and Steel [2003], the set $\mathcal{O}_j$ equals the cluster collection of a uniquely defined tree for each $j \in \{1, 2, \ldots, k\}$. Define R_j to be the tree with $\mathcal{C}(R_j) = \mathcal{O}_j$. Clearly, $R_1 = T_1$. To obtain R_j for any $j \in \{2, \ldots, k\}$, we shall use the following recursive formulation:

LEMMA 4.2. *Let $j \in \{2, \ldots, k\}$ and $A =$ `One-Way_Compatible`(R_{j-1}, T_j). Then* `Merge_Trees`(A, T_j) *is equal to the tree R_j.*

PROOF. By definition, $\mathcal{C}(R_{j-1}) = \mathcal{O}_{j-1}$, and $A =$ `One-Way_Compatible`(R_{j-1}, T_j) is a tree whose cluster collection $\mathcal{C}(A)$ is the subset of $\mathcal{C}(R_{j-1})$ consisting of those clusters that are also compatible with T_j. Thus, $\mathcal{C}(A) = \bigcup_{i=1}^{j-1} \{C \in \mathcal{C}(T_i) : C \text{ is compatible with all trees in } \{T_i, \ldots, T_j\}\}$.

Consequently, `Merge_Trees`(A, T_j) returns a tree whose cluster collection is equal to $\mathcal{C}(A) \cup \mathcal{C}(T_j)$. Trivially, all clusters occurring in T_j are compatible with T_j, so $\mathcal{C}(A) \cup \mathcal{C}(T_j) = \bigcup_{i=1}^{j} \{C \in \mathcal{C}(T_i) : C \text{ is compatible with all trees in } \{T_i, \ldots, T_j\}\} = \mathcal{O}_j$. Hence, this tree is equal to R_j. □

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

Algorithm `Loose_Cons_Tree`
Input: A set $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$ of trees with $\Lambda(T_1) = \Lambda(T_2) = \cdots = \Lambda(T_k)$.
Output: The loose consensus tree of $\mathcal{S}$.

1 $R_1 := T_1$
2 **for** $j := 2$ **to** k **do**
  $A :=$ `One-Way_Compatible`(R_{j-1}, T_j)
  $R_j :=$ `Merge_Trees`(A, T_j)
3 $T := R_k$
4 **for** $j := 1$ **to** k **do**
  $T :=$ `One-Way_Compatible`(T, T_j)
5 **return** T
End `Loose_Cons_Tree`

Fig. 5. Algorithm `Loose_Cons_Tree` for constructing the loose consensus tree.

LEMMA 4.3. *Let T be the loose consensus tree of $\mathcal{S}$. Every cluster that occurs in T also occurs in R_k.*

PROOF. Let C be any cluster in $\mathcal{C}(T)$. By the definition of the loose consensus tree, $C \in \mathcal{C}(T_j)$ for some $j \in \{1, 2, \ldots, k\}$ and C is compatible with all trees in $\{T_1, T_2, \ldots, T_k\}$. In particular, C is compatible with the trees $\{T_j, \ldots, T_k\}$, so $C \in \mathcal{O}_k$, that is, C occurs in R_k. □

As suggested by Lemma 4.3, one strategy for computing the loose consensus tree of $\mathcal{S}$ is to build the tree R_k and then remove certain clusters from it. The next lemma tells us which ones.

LEMMA 4.4. *Let T be the loose consensus tree of $\mathcal{S}$. Then $\mathcal{C}(T) = \{C \in \mathcal{C}(R_k) : C$ is compatible with all trees in $\mathcal{S}\}$.*

PROOF. Consider any $C \in \mathcal{C}(R_k)$. Then for some $j \in \{1, 2, \ldots, k\}$, $C \in \mathcal{C}(T_j)$ and C is compatible with all trees in $\{T_j, \ldots, T_k\}$. If C is also compatible with all trees in $\{T_1, T_2, \ldots, T_k\}$, then C belongs to the set $\{C \in \bigcup_{i=1}^{k} \mathcal{C}(T_i) : C$ is compatible with all trees in $\{T_1, T_2, \ldots, T_k\}\}$, which is equal to $\mathcal{C}(T)$ by the definition of the loose consensus tree.
For the other direction, consider any $C \in \mathcal{C}(T)$. Then $C \in \mathcal{C}(R_k)$ by Lemma 4.3, and C is compatible with all trees in $\{T_1, T_2, \ldots, T_k\}$ by the definition of the loose consensus tree. □

Algorithm `Loose_Cons_Tree` is shown in Figure 5. Its correctness follows from Lemmas 4.3 and 4.4. To analyze its time complexity, observe that every execution of `One-Way_Compatible` takes $O(n)$ time according to Theorem 2.8 and every execution of `Merge_Trees` takes $O(n)$ time by Theorem 2.6, so Step 2 takes $O(nk)$ time. For the same reason, Step 4 takes $O(nk)$ time. We have:

THEOREM 4.5. *Algorithm* `Loose_Cons_Tree` *constructs the loose consensus tree of $\mathcal{S}$ in $O(nk)$ time.*

5. CONSTRUCTING A GREEDY CONSENSUS TREE

We now give an algorithm for building a greedy consensus tree of $\mathcal{S}$ in $O(nq) = O(n^2k)$ time. Recall that p is the number of different clusters and q the total number of clusters occurring in $\mathcal{S}$, with repetitions.

The method in the definition of a greedy consensus tree in Section 1.1 (see also Section 2.1.4 in Bryant [2003]) immediately yields a time complexity of $O(nq + n^2p) = O(n^3k)$. Our improvement comes from eliminating one of the bottlenecks: Instead of

Algorithm `Greedy_Cons_Tree`

Input: A set $\mathcal{S} = \{T_1, T_2, \ldots, T_k\}$ of trees with $\Lambda(T_1) = \Lambda(T_2) = \cdots = \Lambda(T_k) = L$.

Output: A greedy consensus tree of $\mathcal{S}$.

1 Fix an arbitrary ordering of L. For each $T_j \in \mathcal{S}$, compute the bit vector D_j^u of length n for each node u.
2 Sort all the D_j^u-vectors (q in total) 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 ith cluster of the list $\mathcal{X}$ is non-trivial then try to insert it into $\mathcal{C}(T)$ as described in Lemma 5.1.
6 **return** T
End `Greedy_Cons_Tree`

Fig. 6. Algorithm `Greedy_Cons_Tree` for constructing a greedy consensus tree.

first building a maximal set $\mathcal{Y}$ of pairwise compatible clusters in $O(n^2 p)$ time and then constructing a tree T from $\mathcal{Y}$, we build T directly by inserting one cluster at a time.[5] To do so, we use an $O(n)$-time method made possible by Theorem 4.5:

LEMMA 5.1. *For any tree T and $C \subseteq \Lambda(T)$ with $C \notin \mathcal{C}(T)$, it is possible to determine if C is compatible with T and, if so, insert C into $\mathcal{C}(T)$ 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. Clearly, the only non-trivial cluster occurring in T' is C. Let T_{loose} be the loose consensus tree of $\{T, T'\}$. By definition, C is compatible with T if and only if $\mathcal{C}(T_{loose}) = \mathcal{C}(T) \cup \{C\}$ and $|\mathcal{C}(T_{loose})| = |\mathcal{C}(T)| + 1$. Run Algorithm `Loose_Cons_Tree` on $\{T, T'\}$, which takes $O(n)$ time according to Theorem 4.5, and let T_{loose} be its output. If the number of nodes in T_{loose} is larger than that of T (i.e., if the cluster C has been inserted), then let $T := T_{loose}$; otherwise, answer "C is not compatible with T." □

The algorithm is named `Greedy_Cons_Tree` and is listed in Figure 6. Step 1 fixes an arbitrary ordering of L and creates a bit vector D_j^u of length n for each node u in each tree T_j that indicates which leaves belong to $\Lambda(T_j[u])$ (for every $b \in \{1, 2, \ldots, n\}$; the bth bit of D_j^u is set to 1 if and only if the leaf with number b in the ordering is a descendant of u in T_j). Step 2 puts the resulting bit vectors (q in total) in a list $\mathcal{W}$, sorts $\mathcal{W}$, and does a single scan of $\mathcal{W}$ to find the p different clusters in $\mathcal{S}$ and the number of occurrences of each one. Steps 1 and 2 take $O(nq)$ time by doing a bottom-up traversal of each T_i and using radix sort to sort $\mathcal{W}$. Next, Step 3 sorts p integers belonging to $\{1, 2, \ldots, k\}$ to obtain a list $\mathcal{X}$ of all clusters in $\mathcal{S}$ sorted according to frequency, which takes $O(k + p)$ time with counting sort. Finally, Steps 4 and 5 build a greedy consensus tree T by trying to insert each cluster, according to the order in $\mathcal{X}$, into T with the method from Lemma 5.1. Step 4 uses $O(n)$ time, and Step 5 $O(np)$ time because of Lemma 5.1. The total time complexity is $O(nq + k + p + n + np) = O(nq)$. In summary:

THEOREM 5.2. *Algorithm* `Greedy_Cons_Tree` *constructs a greedy consensus tree of $\mathcal{S}$ in $O(nq)$ time.*

[5]Note that Algorithm `Maj_Rule_Cons_Tree` in Section 3 uses a similar kind of strategy; to keep track of candidate clusters efficiently, it stores them in a tree instead of in a list.

Remark 5.3. The problem of determining if T_i and T_j are compatible for all $T_i, T_j \in \mathcal{S}$ and, if so, constructing a tree T' with $\mathcal{C}(T') = \bigcup_{j=1}^{k} \mathcal{C}(T_j)$, can be solved in $O(nk)$ time [Warnow 1994]. Another solution is obtained by adapting the method in the proof of Lemma 5.1 as follows. Let $T' :=$ `Loose_Cons_Tree`$(\mathcal{S})$. For $j \in \{1, 2, \ldots, k\}$, run Day's algorithm (see Section 2.1) with $T_{ref} = T'$ and $T = T_j$. If $\Lambda(T_j[u]) \in \mathcal{C}(T')$ for every $u \in V(T_j)$ and $j \in \{1, 2, \ldots, k\}$, then output T'; otherwise, output "no." The total running time is $O(nk)$.

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 datasets of varying sizes and compared their running times to those of some freely available, widely used software: PHYLIP [Felsenstein 2005], SumTrees in DendroPy [Sukumaran and Holder 2010], and COMPONENT [Page 1993]. (We did not compare our methods to PAUP* [Swofford 2003] because it is commercial software that we did not have access to.) The results are reported in Section 6.2.

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

`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

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 [Bender and Farach-Colton 2000; Harel and Tarjan 1984] was used in the descriptions of the procedures `Merge_Trees` in Section 2.4 and `One-Way_Compatible` in Section 2.5. 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 such a data structure for answering *lca*-queries is presented below.

We use the same notation as in Sections 2.4 and 2.5. For any node $u \in V(T_1)$, let $a := start(u), b := stop(u)$, and let a' and b' be the leaves in L such that $leaf\_rank_{T_2}(a') = a$ and $leaf\_rank_{T_2}(b') = b$. Referring back to Lemma 2.7, 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. We bypass this issue by making use of the correctness of Lemma 2.7 to deduce that $\Lambda(T_1[u])$ is compatible with T_2 if and only if:

- $depth(parent(d_u)) \leq depth(r_u)$ and $depth(parent(e_u)) \leq depth(r_u)$.
- The path from a' to $parent(a'_{left})$ and the path from b' to $parent(b'_{right})$ intersect and therefore share at least one common internal node.
- The internal node common to these two paths which has the greatest depth is $lca(a', b')$.

We construct and store these paths explicitly during the preprocessing phase. For each leaf x, we store the path from x to x_{left} in $left\_path(x)$ and the path from x to x_{right} in

right_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 $depth(a'_{left}) \geq depth(b'_{right})$. We query *right_path*$(b')$ for the node on the path from b' to b'_{right} that is at depth $depth(a'_{left})$. Let $p_1 := a'_{left}$ and $p_2 :=$ the corresponding node on the path from b' to b'_{right}. There are two possibilities:

- If $p_1 = p_2$, then p_1 is the *lca* of a' and b', that is, $r_u = p_1$. From this, we deduce that d_u is the node on *left_path*(a') at depth $depth(a'_{left})+1$, and e_u is the node in *right_path*(b') at the same depth.
- If $p_1 \neq p_2$ and $parent(p_1) = parent(p_2)$, then $parent(p_1) = lca(a', b') = r_u$. Therefore, $p_1 = d_u$ and $p_2 = e_u$.

After finding r_u, d_u, and e_u in this way, the procedures `Merge_Trees` and `One-Way_Compatible` continue their execution as described in Section 2.4 and 2.5.

Greedy consensus tree:

Step 5 of Algorithm `Greedy_Cons_Tree` in Section 5 tries to insert all clusters of $\mathcal{S}$ into the current tree T, one after another according to their frequencies. Lemma 5.1 in Section 5 demonstrated how to do this for any given cluster in $O(n)$ time by applying Algorithm `Loose_Cons_Tree` from Section 4. But since we only need to check if a cluster (rather than an entire tree) is compatible with T, the following 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 $num(u)$ denote this number. To compute $num(u)$, use the formula $num(u) = \sum num(c_i)$ 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 $num(u) = |C|$ is the lowest common ancestor of C in T. Now, determine if C is compatible with $T[u]$ by checking if $num(c_i) = |\Lambda(T[c_i])|$ or 0 for every child c_i of u. This takes $O(n)$ time and the correctness follows from Lemma 2.2.

If C is compatible with T, then insert it as follows: Let $u = lca^T(C)$ 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 $num(c_i) = |\Lambda(T[c_i])|$ 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 `Greedy_Cons_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 ℓ 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 $O(\frac{nq}{\ell}) = O(\frac{n^2k}{\ell})$ time.

6.2. Experimental Results

Simulated datasets:

For certain specified values of n and k, we generated a dataset 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 $\mathcal{S}$ 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 = 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 [Felsenstein 2005]. 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 [Sukumaran and Holder 2010]. The documentation for the implemented algorithm was unavailable.
- **M-Naïve**: A self-implemented, naive algorithm for computing the majority rule consensus tree, based on Wareham [1985]. Given $\mathcal{S}$, it runs Day's algorithm (see Section 2.1) $O(k^2)$ times, using each tree in $\mathcal{S}$ as the reference tree T_{ref} and comparing it against the other trees in $\mathcal{S}$ to count the occurrences of all clusters. A consensus tree is constructed from those clusters that appear more than $\frac{k}{2}$ times. The time complexity is $O(nk^2)$.
- **M-Fast**: An implementation of our new majority rule consensus tree algorithm `Maj_Rule_Cons_Tree` described in Sections 3 and 6.1. Its time complexity is $O(nk)$.
- **L-Naïve**: A self-implemented, naive 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 of $O(\frac{nq}{\ell} + \frac{p^2n}{\ell} + n^2)$. For this implementation, we set $\ell = 60$.
- **L-Fast**: An implementation of our new loose consensus tree algorithm `Loose_Cons_Tree` described in Sections 4 and 6.1. Its time complexity is $O(nk)$.
- **G-PHYLIP**: The greedy consensus tree method in PHYLIP [Felsenstein 2005]. 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 $O(\frac{n}{\ell})$ by using words of length ℓ. The expected time complexity is $O(q + \frac{n^2q}{\ell} + n^2)$.
- **G-Naïve**: A naive variant of the algorithm used in G-PHYLIP. The difference is that hashing is not used to count the clusters. Instead, words of length $\ell = 60$ are used to speed up the computations. The time complexity is $O(\frac{nq}{\ell} + \frac{nq^2}{\ell} + n^2)$.
- **G-Fast**: An implementation of our new greedy consensus tree algorithm `Greedy_Cons_Tree` described in Sections 5 and 6.1. Its time complexity is $O(\frac{nq}{\ell} + np)$. For this implementation, we set $\ell = 60$.

In addition to the above, the program COMPONENT [Page 1993] was also considered. This software uses hashing to compute its results. However, COMPONENT 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:

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

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

For each of (a)–(d), we generated 10 datasets, applied the methods, and measured their running times. The purpose of case (c) was to demonstrate that M-Fast is much faster than M-PHYLIP when $n \gg k$, and the purpose of case (d) was to investigate the performance of M-Fast and M-PHYLIP for very large inputs. (Thus, we did not run the other methods for (c) and (d).) In (e), we generated at least three datasets 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.

All experiments were carried out on Ubuntu Nutty Narwhal, a 64-bit operating system with 8.00GB RAM and a CPU running at 2.20GHz. The worst-case and average running times (in seconds) are reported below.

Experimental results:

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

	Worst-case	Average
M-PHYLIP	1.94	1.88
M-SumTrees	91.18	89.55
M-Naïve	291.19	274.96
M-Fast	3.72	3.69
L-Naïve	8.00	7.12
L-Fast	5.34	5.16
G-PHYLIP	2.94	2.67
G-Naïve	4.34	4.14
G-Fast	4.10	3.76

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

	Worst-case	Average
M-PHYLIP	34.07	30.03
M-SumTrees	932.12	918.55
M-Naïve	1100.69	1089.57
M-Fast	16.09	14.86
L-Naïve	335.31	319.03
L-Fast	22.11	21.85
G-PHYLIP	67.19	63.09
G-Naïve	115.72	111.78
G-Fast	41.32	40.08

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

	Worst-case	Average
M-PHYLIP	93.25	90.04
M-SumTrees	—	—
M-Naïve	—	—
M-Fast	4.40	4.28
L-Naïve	—	—
L-Fast	—	—
G-PHYLIP	—	—
G-Naïve	—	—
G-Fast	—	—

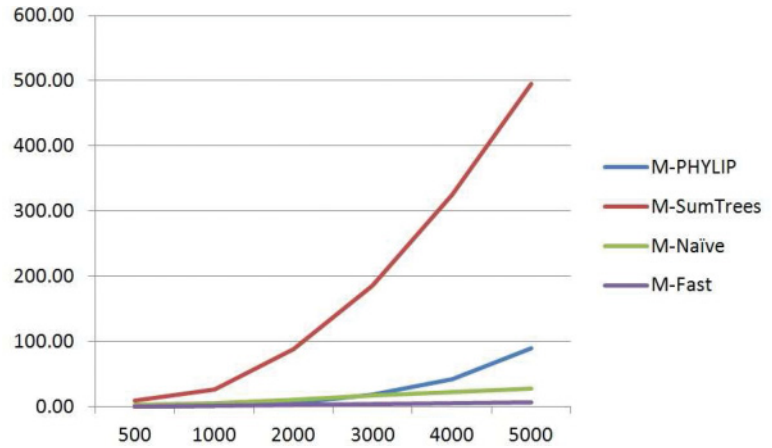
(d) $n = 2000$, $k \in \{2000, 3000, 4000, 5000\}$:

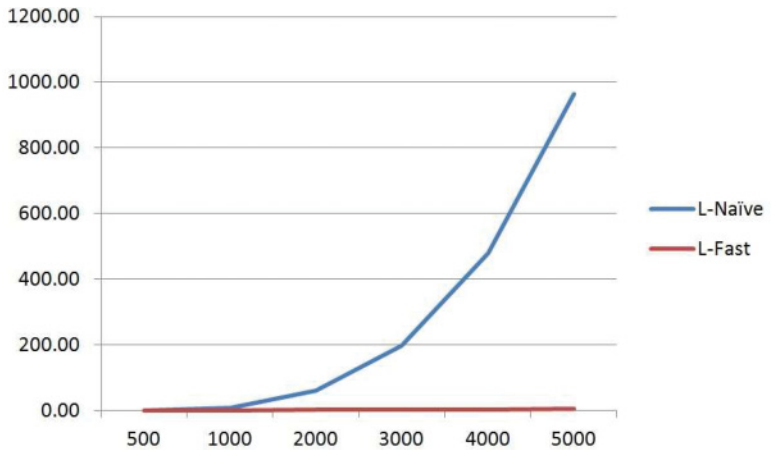
M-PHYLIP returned "Error allocating memory" for $n = 2000$, $k \geq 2000$, whereas M-Fast worked fine and obtained the following worst-case and average running times.

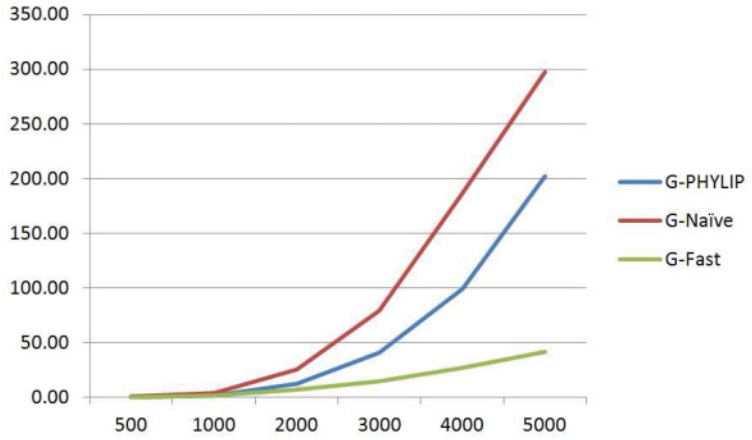
k	Worst-case	Average
2000	31.22	30.86
3000	47.42	46.23
4000	62.54	61.88
5000	78.96	77.78

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

In the next 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.

—M-Fast (`Maj_Rule_Cons_Tree`) was better than SumTrees and COMPONENT for all datasets. Furthermore, it was significantly faster than PHYLIP when n was large and k was small (for $n = 5000$, $k = 100$, it was about 20 times faster). On the other hand, for small n, PHYLIP was faster. This behavior can be explained by the probability of collisions in the hash tables that PHYLIP uses to store clusters increasing as n increases.
—M-Fast (`Maj_Rule_Cons_Tree`) may come in handy when analyzing huge phylogenetic datasets in the future. For large inputs ($n = 2000$ and $2000 \leq k \leq 5000$), PHYLIP ran out of memory but our algorithm did not.
—L-Fast (`Loose_Cons_Tree`) could handle much larger datasets than COMPONENT and ran quickly, producing a solution for the dataset with $n = 2000$, $k = 1000$ in a little over 20s.
—G-Fast (`Greedy_Cons_Tree`) was slower than PHYLIP when n and k were small and $n \ll k$. It outperformed PHYLIP as the datasets got larger and $n \gg k$.

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

7. FINAL REMARKS

To end this article, we briefly mention a few other useful types of consensus trees and some related open problems. As above, 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 of cardinality n.

First, a *strict consensus tree of* $\mathcal{S}$ [Sokal and Rohlf 1981] is a tree T with $\Lambda(T) = L$ containing precisely those clusters that occur in every tree in $\mathcal{S}$, i.e., $\mathcal{C}(T) = \bigcap_{i=1}^{k} \mathcal{C}(T_i)$. This type of consensus tree is well understood [Bryant 2003; Felsenstein 2004; Sung 2010]. The advantages of the strict consensus tree is that it is always unique and can be computed quickly; the algorithm by Day [1985] (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 $\mathcal{S}$, so the strict consensus tree of $\mathcal{S}$ is just a root node to which the leaves a, b, c, d, e are directly attached.

Second, an *R* consensus tree of* $\mathcal{S}$ [Bryant 2003] 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 $\mathcal{R}_{maj}$ and no other rooted triplets; see Bryant [2003], Degnan et al. [2009], and Jansson and Sung [2013] for the definition. The R* consensus tree has several nice properties [Degnan et al. 2009], but it is still not known how to compute it efficiently. The fastest methods run in $O(n^3k)$ time for unbounded k [Bryant 2003; Jansson and Sung 2013] and in $O(n^2\sqrt{\log n})$ time when $k = 2$ [Jansson and Sung 2013], and it is an open problem to reduce their running times.

Third, 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 introduced by Lott et al. [2009] and further studied in Cui et al. [2012] and Huber et al. [2012]. 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. A challenging task is to define informative types of consensus MUL-trees that admit efficient algorithms.

Some other types of consensus trees that we are currently working on are the *Adams consensus tree* [Adams III 1972], the *majority rule (+) consensus tree* [Dong et al. 2010], the *frequency difference consensus tree* [Goloboff et al. 2003], and *local consensus trees* [Kannan et al. 1998]. Any new consensus tree algorithms that we implement will be included in the FACT package (see Section 6). For further discussions on the advantages and disadvantages of different types of consensus trees, see [Bryant 2003; Degnan et al. 2009; Felsenstein 2004; Holder et al. 2008; Sung 2010].

ACKNOWLEDGMENTS

The authors thank Joseph Felsenstein for clarifications regarding PHYLIP and Todd Wareham for confirming the time complexity of the deterministic algorithm in Wareham [1985].

REFERENCES

E. N. Adams III. 1972. Consensus techniques and the comparison of taxonomic trees. *Syst. Zool.* 21, 4 (1972), 390–397.

N. Amenta, F. Clarke, and K. St. John. 2003. A linear-time majority tree algorithm. In *Proceedings of the 3rd International Workshop on Algorithms in Bioinformatics* (WABI 2003). Lecture Notes in Computer Science, Vol. 2812. Springer-Verlag, Berlin, 216–227.

M. S. Bansal, J. Dong, and D. Fernández-Baca. 2011. Comparing and aggregating partially resolved trees. *Theor. Comput. Sci.* 412, 48 (2011), 6634–6652.

M. A. Bender and M. Farach-Colton. 2000. The LCA problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics* (LATIN 2000). Lecture Notes in Computer Science, Vol. 1776. Springer-Verlag, Berlin, 88–94.

R. S. Boyer and J. S. Moore. 1991. MJRTY – a fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, R. S. Boyer (Ed.). Kluwer Academic Publishers, Amsterdam, 105–117.

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

D. Bryant. 2003. 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.). DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 61. American Mathematical Society, Washington, DC, 163–184.

J. A. Cotton and M. Wilkinson. 2007. Majority-rule supertrees. *Syst. Biol.* 56, 3 (2007), 445–452.

Y. Cui, J. Jansson, and W.-K. Sung. 2012. Polynomial-time algorithms for building a consensus MUL-tree. *J. Comput. Biol.* 19, 9 (2012), 1073–1088.

W. H. E. Day. 1985. Optimal algorithms for comparing trees with labeled leaves. *J. Clas.* 2, 1 (1985), 7–28.

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

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

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

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

P. A. Goloboff, J. S. Farris, M. Källersjö, B. Oxelman, M. J. Ramírez, and C. A. Szumik. 2003. Improvements to resampling measures of group support. *Cladistics* 19, 4 (2003), 324–332.

D. Gusfield. 1997. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York.

D. Harel and R. E. Tarjan. 1984. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.* 13, 2 (1984), 338–355.

M. T. Holder, J. Sukumaran, and P. O. Lewis. 2008. A justification for reporting the majority-rule consensus tree in Bayesian phylogenetics. *Syst. Biol.* 57, 5 (2008), 814–821.

K. T. Huber, V. Moulton, A. Spillner, S. Storandt, and R. Suchecki. 2012. Computing a consensus of multilabeled trees. In *Proceedings of the 14th Workshop on Algorithm Engineering and Experiments (ALENEX 2012)*. SIAM, Philadelpia, PA, 84–92.

J. Jansson, C. Shen, and W.-K. Sung. 2013. Improved algorithms for constructing consensus trees. In *Proceedings of the 24thAnnual ACM-SIAM Symposium on Discrete Algorithms (SODA 2013)*. SIAM, Philadelphia, PA, 1800–1813.

J. Jansson and W.-K. Sung. 2013. Constructing the R* consensus tree of two trees in subcubic time. *Algorithmica* 66, 2 (2013), 329–345.

R. J. Jensen. 1983. Report on sixteenth international numerical taxonomy conference. *Syst. Zool.* 32, 1 (1983), 83–89.

S. Kannan, T. Warnow, and S. Yooseph. 1998. Computing the local consensus of trees. *SIAM J. Comput.* 27, 6 (1998), 1695–1724.

M. K. Kuhner and J. Felsenstein. 1994. A simulation comparison of phylogeny algorithms under equal and unequal evolutionary rates. *Molec. Biol. Evol.* 11, 3 (1994), 459–468.

M. Lott, A. Spillner, K. T. Huber, A. Petri, B. Oxelman, and V. Moulton. 2009. Inferring polyploid phylogenies from multiply-labeled gene trees. *BMC Evolution. Biol.* 9 (2009), 216.

T. Margush and F. R. McMorris. 1981. Consensus *n*-trees. *Bull. Math. Biol.* 43, 2 (1981), 239–244.

F. R. McMorris, D. B. Meronk, and D. A. Neumann. 1983. A view of some consensus methods for trees. In *Numerical Taxonomy: Proceedings of the NATO Advanced Study Institute on Numerical Taxonomy*, J. Felsenstein (Ed.). NATO ASI Series, Vol. G1. Springer-Verlag, Berlin, 122–126.

F. R. McMorris and M. Wilkinson. 2011. Conservative supertrees. *Syst. Biol.* 60, 2 (2011), 232–238.

L. Nakhleh, T. Warnow, D. Ringe, and S. N. Evans. 2005. A comparison of phylogenetic reconstruction methods on an Indo-European dataset. *Trans. Philol. Soc.* 103, 2 (2005), 171–192.

R. Page. 1993. COMPONENT, version 2.0. Software package, University of Glasgow, U.K. (1993).

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

C. Semple and M. Steel. 2003. *Phylogenetics*. Oxford Lecture Series in Mathematics and Its Applications, Vol. 24. Oxford University Press, Oxford.

R. R. Sokal and F. J. Rohlf. 1981. Taxonomic congruence in the leptopodomorpha re-examined. *Syst. Zool.* 30, 3 (1981), 309–325.

J. Sukumaran and M. T. Holder. 2010. DendroPy: A python library for phylogenetic computing. *Bioinformatics* 26, 12 (2010), 1569–1571.

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

D. L. Swofford. 2003. PAUP*, version 4.0. Software package, Sinauer Associates, Inc., Sunderland, MA (2003).

H. T. Wareham. 1985. An efficient algorithm for computing M_l consensus trees. B.Sc. Honours thesis, Memorial University of Newfoundland, Canada. (1985).

T. J. Warnow. 1994. Tree compatibility and inferring evolutionary history. *J. Algorith.* 16, 3 (1994), 388–407.

Received March 2013; accepted April 2016