SIAM J. COMPUT.
Vol. 27, No. 6, pp. 1695–1724, December 1998

009

COMPUTING THE LOCAL CONSENSUS OF TREES*

SAMPATH KANNAN†, TANDY WARNOW†, AND SHIBU YOOSEPH†

Abstract. The inference of consensus from a set of evolutionary trees is a fundamental problem in a number of fields such as biology and historical linguistics, and many models for inferring this consensus have been proposed. In this paper we present a model for deriving what we call a *local consensus tree* T from a set of trees $\mathcal{T}$. The model we propose presumes a function f, called a *total local consensus function*, which determines for every triple A of species, the form that the local consensus tree should take on A. We show that all local consensus trees, when they exist, can be constructed in polynomial time and that many fundamental problems can be solved in linear time. We also consider *partial local consensus functions* and study optimization problems under this model. We present linear time algorithms for several variations. Finally we point out that the local consensus approach ties together many previous approaches to constructing consensus trees.

Key words. algorithms, graphs, evolutionary trees

AMS subject classifications. 05C05, 68Q25, 92-08, 92B05

PII. S0097539795287642

1. Introduction.

An *evolutionary tree* (also called a *phylogeny* or *phylogenetic tree*) for a species set S is a rooted tree with $|S| = n$ leaves labeled by distinct elements in S. Because evolutionary history is difficult to determine (it is both computationally difficult as most optimization problems in this area are *NP-hard* and scientifically difficult as well since a range of approaches appropriate to different types of data exist), a common approach to solving this problem is to apply many different algorithms to a given data set, or to different data sets representing the same species set, and then look for common elements from the set of trees which are returned.

There is extensive literature about inferring consensus from ordered sets of trees, with much attention paid to the properties of the rules for inferring the consensus. In this paper, we will make an explicit assumption that the consensus rule be *independent* of the ordering of the trees in the input; i.e., we will presume that the input to the consensus problem is an unordered multiset of evolutionary trees, each leaf-labelled by the elements in S. We call this input a *profile*, noting that in this paper the terminology is restricted in meaning as we have indicated.

Several consensus methods are described in the literature for deriving one tree from a profile of evolutionary trees. These methods include maximum agreement subtrees [16, 19, 13, 24, 14], strict consensus trees [4, 9], median trees (also known as majority trees) [5], compatibility trees [10, 11, 12], the Nelson tree [22], and the Adams consensus [1].

The algorithms for some of these are implemented in standard packages and are in use; most common, perhaps, are strict and majority consensus tree approaches.

*Received by the editors June 8, 1995; accepted for publication (in revised form) September 12, 1996; published electronically June 3, 1998. The research of the first author was supported in part by NSF grant CCR-9108969. The research of the second author was supported in part by ARO grant DAAL03-89-0031PRI, NSF Young Investigator Award, and by generous support from Paul Angello. The research of the third author was supported in part by ARO grant DAAL03-89-0031PRI, a fellowship from the Institute for Research in Cognitive Science at the University of Pennsylvania, and a fellowship from the Program in Mathematics and Molecular Biology at the University of California at Berkeley, which is supported by NSF grant DMS-9406348.

http://www.siam.org/journals/sicomp/27-6/28764.html

†Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104 (kannan@central.cis.upenn.edu, tandy@central.cis.upenn.edu, yooseph@saul.cis.upenn.edu).

One notion of the *information* content of an evolutionary tree is the degree of resolution indicated by the tree; this can be quantified in a number of ways, for example, by counting the number of internal nodes or the number of resolved triples[1] in the tree. This is because the most usual interpretation of an unresolved triple in an evolutionary tree is that the evolutionary history of that triple cannot be absolutely inferred from the data. Thus, for example, a completely resolved tree (i.e., a binary tree) asserts a hypothesis about the evolution of all triples of taxa, while the star (i.e., root with all taxa children of the root) does not assert any hypothesis about the evolution of any triple. One of the motivations for proposing this new model of consensus tree construction is the observation that on some data sets the strict and majority consensus trees may be fairly uninformative (i.e., be fairly unresolved).

In this paper, we propose a new model, called the *local consensus*. This model is based upon functions, called *local consensus functions*, for inferring the rooted topology of the homeomorphic subtree induced by triples of species. We will show that given any local consensus function, we can determine whether a tree (called the *local consensus tree*) consistent with the constraints implied by the local consensus function can be computed in polynomial time and that many of the natural forms of the local consensus can be computed in linear time. We also analyze optimization problems based upon partial local consensus rules and show that many of these can also be solved in polynomial time. We will show that this method unifies many of the previously favored approaches while providing greater flexibility to the biologists in the interpretation of the data. Furthermore, the local consensus trees produced are, in most cases, significantly more informative (in the sense of more refined; see the above discussion) than trees produced using the strict or majority consensus methods.

2. Preliminaries.

2.1. Trees. Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of species. An *evolutionary tree* for S (also known as a phylogenetic tree or, more simply, a phylogeny) is a rooted tree T with n leaves each labeled by a distinct element from S. The internal nodes denote ancestors of the species in S. For an arbitrary subset $S' \subset S$ we denote by $\mathbf{T}|\mathbf{S'}$ the homeomorphic subtree of T induced by the leaves in S'. In particular, for a specified triple $\{a, b, c\} \subset S$ we denote by $\mathbf{T}|\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ the homeomorphic subtree of T induced by the leaves labeled by a, b, and c. This topology is completely determined by specifying the pair of species among a, b, and c whose least common ancestor (LCA) lies farthest away from the root. If (a, b) is this pair then we denote this by $((a, b), c)$, and T is said to be *resolved* on the triple a, b, c. If T is not binary it may happen that all three pairs of species have the same LCA. In this case we will say that a, b, c is *unresolved* in T and denote this topology by (a, b, c). In this paper, when we say a triple a, b, c is resolved, we mean that $T|\{a, b, c\}$ is one of $((a, b), c), ((a, c), b)$, or $((b, c), a)$.

For a *profile* P, which is defined by a multiset $\{T_1, T_2, \ldots, T_k\}$, we let $P|\{a, b, c\}$ denote the multiset $\{T_1|\{a, b, c\}, T_2|\{a, b, c\}, \ldots, T_k|\{a, b, c\}\}$.

Given a tree T containing nodes u, v, w, we let $lca_T(u, v, w)$ denote the LCA of u, v, and w in T. Also, we let $u \leq_T v$ denote that v is on the path from u to the root of T.

2.2. Local consensus functions, rules, and trees. Let $\mathcal{T}(a, b, c)$ denote the set of rooted subtrees on the leaf set $\{a, b, c\} \subseteq S$; thus $|\mathcal{T}(a, b, c)| = 4$, with three of

[1] See section 2.1 for definitions of a *resolved* triple and an *unresolved* triple.

the trees being resolved and one being the star (i.e., unresolved) tree on a, b, c.

A *local consensus function* is a function f which specifies the constraints for certain (i.e., perhaps not all) triples a, b, c of species. Let A be the set of all three element subsets of S. We define $f : A \to \cup_{\{a,b,c\} \in A} \mathcal{T}(a, b, c) \cup \{*\}$. When $f(X) = *$, for some $X = \{a, b, c\} \in A$, this indicates that the form of the triple a, b, c is *unconstrained*. When $f(X) \neq *\forall X \in A$, i.e., no triple is unconstrained, then f is said to be a *total* local consensus function. Otherwise, f is said to be a *partial* local consensus function.

A rooted tree T (if it exists) which is leaf-labelled by elements from S and which meets all the constraints implied by the local consensus function f is called an *f-local consensus tree*.[2] Note that when a triple a, b, c is set to be unconstrained by f, then $T|\{a, b, c\}$ can be any of the elements in $\mathcal{T}(a, b, c)$. Thus T is a tree such that for all triples $X \in A, T|X = f(X)$, if $f(X) \neq *$.

A local consensus function can be applied to a profile P. It is also possible for the local consensus function to define the form of the output triple based upon the forms the triple takes in the profile. Such local consensus functions are called *local consensus rules*. Let $\mathcal{M}$ be the set of all multisets of size k, where each element of a multiset belongs to $\mathcal{T}(a, b, c)$. A local consensus rule is a function $f : \mathcal{M} \to \mathcal{T}(a, b, c) \cup \{*\}$. If $f(X) = *$, for some $X \in \mathcal{M}$, then f is said to be a partial local consensus rule; otherwise, f is a total local consensus rule.

Given a profile P and a local consensus rule f, the f-local consensus tree (if it exists) is a rooted tree T such that for all triples $X \subseteq S, T|X = f(P|X)$, if $f(P|X) \neq *$.[3]

It is not the case that a local consensus tree necessarily exists for an arbitrary local consensus function (or rule) applied to an arbitrary input profile. Determining whether a local consensus tree exists, and constructing it when it does, is the subject of this paper.

The structure of the paper is as follows. In section 3, we will describe some general techniques for determining if a local consensus tree exists. In particular, we will give a polynomial time algorithm (based upon the algorithm in [3]), which can determine if a local consensus tree exists for an arbitrary local consensus function (or rule), and construct it when it does. We will also describe a class of natural local consensus rules and describe general techniques for constructing local consensus trees from such natural local consensus rules when they exist. In section 4, we then describe some specific natural local consensus rules and some fast algorithms for constructing the local consensus trees. In section 5, we consider optimization problems related to constructing local consensus trees and present efficient algorithms to solve some of these optimization problems. We conclude in section 6 with a discussion and suggestions for extensions.

3. Techniques.

3.1. General local consensus functions. For an arbitrary local consensus function f and an arbitrary profile of trees $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$, we can compute the constraint indicated by f for every triple of species a, b, c. This produces a set of $O(n^3)$ constraints on the consensus tree we wish to construct, where each constraint is a rooted tree for a triple on a species set a, b, c. This rooted tree may be resolved

[2] We will also sometimes refer to it simply as a local consensus tree.

[3] Note that f is defined the same on all triples $X \subseteq S$. As defined above, the triple labels a, b, c serve merely as place holders. The definition of a local consensus rule can easily be changed to accommodate a *different* rule for each triple.

(i.e., it may be of the form $((a,b)c)$) or it may be unresolved (i.e., of the form (a,b,c)). If there is a tree T meeting all these constraints, then T is the local consensus tree for f. Thus, we can reduce the problem of consensus tree construction for an arbitrary local consensus function to the problem of determining consistency of a set of rooted triples.

3.1.1. Rooted triple consistency. We present results related to this general problem.

THEOREM 3.1. *Determining if a tree T exists which meets a set of constraints (and constructing it if it does) can be solved in $O(pn \log n)$ time if the constraints include unresolved triples and otherwise can be solved in $O(pn)$ time, where p is the number of constraints defined by f.*

Proof. In [3], Aho et al. describe algorithms which determine if a family of constraints on LCA relations can be satisfied within a single rooted tree. We describe here the simple algorithm they give for the case where the constraints are given as rooted resolved triples $((x,y),z)$. For such input the algorithm works top-down figuring out the clusters at the children of the root before recursing. To do this the algorithm maintains disjoint sets. Initially all leaves are in singleton sets. For each rooted triple $((x,y),z)$ the algorithm unions the sets containing x and y to indicate that x and y must lie below the same child of the root. This algorithm never unions sets unless this is forced. Recursive calls include constraints that are on species entirely contained in the same component discovered in the previous call. If all the species are seen to be in the same component (either initially or during a recursive call), the algorithm determines that the constraints cannot be simultaneously satisfied. This simple algorithm has a worst-case behavior of $O(pn)$, where there are p LCA constraints and the underlying set S has n elements which will be leaves in the final tree. ∎

However, we can also solve the consistency problem faster than by using the Aho et al. algorithm. In [21], an algorithm is given for the problem addressed in [3] for the case where all the triples are resolved. In this case a faster algorithm can be obtained.

LEMMA 3.1 (Henzinger, King, and Warnow [21]). *Let $\mathcal{A}$ be a set of p resolved rooted triples on a leaf set S with $|S| = n$. We can determine in $\min\{O(p\sqrt{n}), O(p + n^{2.5})\}$ time whether a tree T exists such that $T|\{a,b,c\}$ is homeomorphic to the rooted triple(s) in A on $\{a,b,c\}$ (if such a triple exists in A).*

In the context of the rooted triple consistency problem, we also refer to the work of [8, 7], where the conditions necessary for a given set of triple constraints to define a tree are investigated.

3.2. Constructing local consensus trees in polynomial time. As a consequence of the results in the previous section, we can prove the following theorem.

THEOREM 3.2. *Let f be an arbitrary partial local consensus rule and $\mathcal{T}$ a set of k evolutionary trees on S with $|S| = n$.*

1. *If every triple which is not set to $*$ is defined to be resolved by f, then we can determine if the local consensus tree exists and construct it if it does in $O(kn^3)$ time.*
2. *If f defines some triples (which are not set to $*$) to be unresolved, then we can determine if the local consensus tree exists and construct it if it does in $O(kn^3 + n^4 \log n)$ time.*

Proof. Given f, $\mathcal{T}$, and a triple A, we can determine the form of $\mathcal{T}_f|A$ (for those triples A for which $T_f|A$ is not unconstrained) in $O(kn^3)$ time. If all the triples which are not set to unconstrained are defined to be resolved, then by Lemma 3.1 we can determine if the partial local consensus tree exists and construct it if it does, in

$O(n^{2.5} + p)$ time, where p is the number of constraints. The total time is therefore bounded by the cost of computing the triples. If some of the triples are unresolved then we can use Theorem 3.1 to get an $O(kn^3 + n^4 \log n)$ algorithm which will determine if the tree exists and construct it when it does. □

3.2.1. Constructing local consensus trees from total local consensus rules. While local consensus trees can be constructed in $O(kn^3)$ time from partial local consensus rules, local consensus trees can be computed even faster when the local consensus rule is *total*.

LEMMA 3.2 (Kannan, Lawler, and Warnow [18]). *Given an oracle $\mathcal{O}$ which can answer queries of* "What is the form of $T|\{a, b, c\}$ for a species set $\{a, b, c\}$?"*, we can construct in $O(n^2)$ time a tree T consistent with all the oracle queries (if it exists) and $O(rn \log n)$ time if the tree T has degree bounded by r.*

THEOREM 3.3. *Let f be a total local consensus rule. Then given a set of k rooted trees on n species, we can construct in $O(kn^2)$ time the f-local consensus tree $\mathcal{T}_f$ if it exists. If f always returns resolved subtrees, then we can compute $\mathcal{T}_f$ in $O(kn \log n)$ time.*

Proof. We can implement the oracle determining the form of the homeomorphic subtree of $\mathcal{T}_f$ on a triple a, b, c by first preprocessing the trees to answer LCA queries in constant time using [20]. Then, answering a query needs only $O(k)$ time. By [18], we need only $O(n^2)$ queries and $O(n^2)$ additional work for a total cost of $O(kn^2)$ in the general case. When $\mathcal{T}_f$ has degree bounded by r, we have total cost $O(krn \log n)$. If f always returns resolved subtrees, then $\mathcal{T}_f$ will be binary, so that the total cost is $O(kn \log n)$. □

Note, however, that this algorithm does not verify that the tree constructed is the local consensus tree; that is, it is possible that the constraints are inconsistent, so that no local consensus tree exists for that local consensus function (or rule). When it does, however, the tree constructed will equal the local consensus tree. Thus, when it can be shown that the local consensus tree *does* exist, then this method will necessarily produce the local consensus tree. In general, however, it will be necessary to verify that the constructed tree is the local consensus tree.

We have described two algorithms for inferring whether a local consensus tree exists for an arbitrary local consensus function (or rule). When the local consensus function (or rule) is total, if the local consensus tree exists, it can be constructed in $O(kn^2)$ time, where k is the number of trees in the profile and n is the number of leaves in each tree. However, the tree that results then needs to be verified to be the local consensus tree (and the fastest verification algorithm may still require $\Omega(kn^3)$ time). When the local consensus function (or rule) is partial, then a slower $O(kn^3)$ algorithm can be used, but it simultaneously constructs and verifies that the constructed tree is the local consensus tree.

3.3. Local consensus rules. A local consensus rule must handle essentially three types of situations for each pattern of subtrees in the profile for a triple a, b, c of species: *profile constant on a,b,c; profile compatible on a,b,c; profile incompatible on a,b,c.* The profile of trees may agree on that set a, b, c, and thus all reflect the same evolutionary history, or the trees may differ (in two different ways) on the triple. Depending upon the pattern of different subtrees, the local consensus rule may elect to constrain the form of the output or to leave the output unconstrained for that triple. However, we will only consider a local consensus rule to be natural if it is *conservative*, where by conservative we mean the following definition.

DEFINITION 3.1. *Let P be a profile of evolutionary trees and f be a local consensus rule. Then f is said to be* conservative *for every triple a, b, c, iff, $f(P|\{a, b, c\}) = ((a, b), c)$, then a, b, c is not resolved as $((a, c), b)$ or $((b, c), a)$ in any of the trees in P.*

Being conservative is obviously a natural requirement, since to enforce a topological constraint which is contradicted in the profile is clearly unmotivated.

We now describe the three general scenarios that may arise and discuss the possible constraints that may arise under natural local consensus rules.

Profile constant on a, b, c. If all the trees in the profile have the same form on a triple a, b, c, then we say the profile is *constant* on a, b, c. In this case, a natural local consensus rule should either require that the consensus tree have the same form as the trees in the profile, or it may leave the form unconstrained.

Profile compatible on a, b, c. If all the trees in the profile that have resolved subtrees for a, b, c have the *same* resolved form (i.e., no two trees in the profile resolve a, b, c differently), then the profile is said to be *compatible* on a, b, c. In this case, the natural local consensus rule may elect to leave the tree unconstrained for a, b, c; otherwise, it should constrain the output to either be the unique resolution indicated by the profile or should constrain it to be *unresolved*. In the first case, we call the local consensus rule *optimistic*, and in the second case we call the local consensus rule *pessimistic*.

Profile incompatible on a, b, c. The remaining case is where the profile contains trees which have different resolutions for a, b, c. In this case, a natural local consensus rule may elect to require the consensus tree to be unresolved, or it may select one of the resolutions represented in the profile[4] (perhaps selecting the resolution with the plurality representation), or it may not constrain the output at all.

A local consensus rule can be defined by deciding how it will respond to each of the different situations that can arise. Thus, for example, a natural local consensus rule may require that when the profile is constant on a, b, c, then the output tree is constrained to have that same form, and it may elect to be optimistic in the presence of compatible forms on a, b, c but may leave unconstrained any triple for which the profile is incompatible.

In all of our following discussions, we restrict ourselves to profiles of two trees. The techniques and most observations can be generalized.

4. Specific total local consensus rules. As examples of natural local consensus rules, we will define two total local consensus rules: the *optimistic local consensus (OLC) rule* and the *pessimistic local consensus (PLC) rule*. These are not the only natural local consensus rules that are worthy of study, but the techniques used for constructing local consensus trees for these rules are indicative of general approaches for greatly speeding up the construction and verification phases used in the previous section.

When the trees are not necessarily binary, the local consensus rule may encounter triples for which the profile is not constant but is nevertheless compatible. Because a total local consensus rule must constrain the form of each triple for the consensus tree, it must determine whether to require that the rooted triple be resolved or unresolved. This decision is based upon the interpretation of an unresolved triple, which can be made in one of two ways: *any resolution of the three-way split is possible* or *the unresolved triple indicates a three-way speciation event*. If the local consensus rule

[4]In this case the *conservative* nature of the rule need not be maintained.

chooses to interpret lack of resolution as being consistent with any resolution, then it will constrain the output to be resolved according to the unique resolution present in the profile, and otherwise it will constrain the output to be unresolved. The first type of total local consensus rule is said to be *optimistic* and the second type *pessimistic*.

We now define these two consensus rules.

DEFINITION 4.1. *Let T_1 and T_2 be two rooted trees on the same leaf set S. A rooted tree T is called the* OLC *of T_1 and T_2 iff for each triple a, b, c, $T|\{a, b, c\} = ((a, b), c)$ iff $T_i|\{a, b, c\} = ((a, b), c)$ and $T_j|\{a, b, c\} = ((a, b), c)$ or (a, b, c) for $\{i, j\} = \{1, 2\}$.*

DEFINITION 4.2. *Let T_1 and T_2 be two rooted trees on the same leaf set S. A rooted tree T is called the* PLC *of T_1 and T_2 iff for each triple a, b, c, $T|\{a, b, c\} = ((a, b), c)$ iff $T_1|\{a, b, c\} = T_2|\{a, b, c\} = ((a, b), c)$.*

In the next two subsections we discuss efficient algorithms for these rules. But first we give some basic and standard definitions.

DEFINITION 4.3. *Let T be a rooted tree with leaf set S. Given a node $v \in V(T)$, we denote by $\mathcal{L}(T_v)$ the set of leaves in the subtree T_v of T rooted at v. This is also called the cluster at v and is represented by α_v. The set $C(T) = \{\alpha_v : v \in V(T)\}$ is called the* cluster encoding of T.

Every rooted tree in which the leaves are labeled by S contains all singletons and the entire set S in $C(T)$; these clusters are called the *trivial clusters*. We define a *maximal* cluster to be the cluster defined by the child of the root. (Here we allow for a maximal cluster to be defined by a leaf also.)

We also define the notion of compatibility of a set of clusters.

DEFINITION 4.4. *A set A of clusters is said to be* compatible *iff there exists a tree T such that $C(T) = A$.*

The following proposition can be found in [17].

PROPOSITION 4.1. *A set A of clusters is compatible iff $\forall \alpha_i, \alpha_j \in A$, $\alpha_i \cap \alpha_j \in \{\alpha_i, \alpha_j, \emptyset\}$.*

We now state a theorem which will be used in the later sections.

THEOREM 4.1. *Let T_1 and T_2 be two rooted trees on the same leaf set S and let f be a conservative local consensus rule. If the f-local consensus tree T exists, then $C(T) \cup C(T_1)$ and $C(T) \cup C(T_2)$ are compatible sets.*

Proof. Suppose not and suppose without loss of generality that $C(T) \cup C(T_1)$ is not a compatible set. Then by Proposition 4.1, $\exists \alpha \in C(T)$ and $\beta \in C(T_1)$ such that $\alpha \cap \beta \notin \{\alpha, \beta, \emptyset\}$. Pick $a \in \alpha \cap \beta$, $b \in \alpha - \beta$ and $c \in \beta - \alpha$. The topology of the triple a, b, c in T_1 is $((a, c), b)$ while in T it is $((a, b), c)$. Since f is a conservative local consensus rule, this is impossible. ∎

4.1. OLC. In this section we look at the problem of finding the OLC tree of two trees defined in the previous section. Note that the OLC of two trees may not exist. See Figure 1 for an example.

4.1.1. Characterization of the OLC tree. The following lemma characterizes the OLC tree when it exists.

THEOREM 4.2. *Let T_1 and T_2 be two rooted trees on the same species set S. If the OLC tree T_{olc} exists, then $C(T_{olc}) = A$, where $A = \{\alpha^* \mid \alpha^* = \alpha_1 \cap \alpha_2$, where $\alpha_1 \in C(T_1)$ and $\alpha_2 \in C(T_2)$, and α^* is compatible with both $C(T_1)$ and $C(T_2)\}$.*

Proof. Pick any cluster $\alpha \in A$. If we look at any triple x, y, z with $x, y \in \alpha$ and $z \notin \alpha$, then this triple will be resolved as $((x, y), z)$ in one tree and will be either resolved the same or unresolved in the other tree. In either case, $\alpha \in C(T_{olc})$.

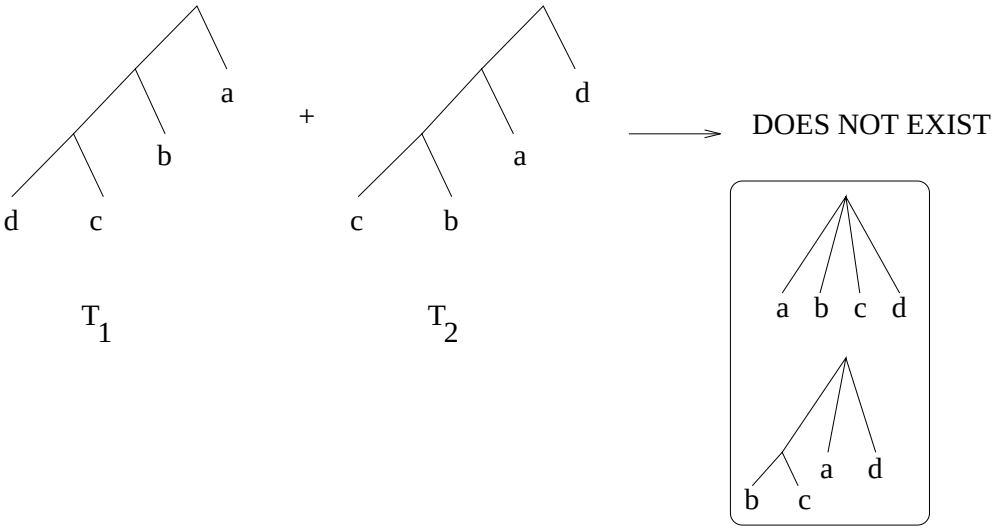

FIG. 1. *Example showing that the OLC need not always exist. The trees in the box are possible candidates, but they each fail to maintain the necessary topology for some triple.*

Conversely, pick any cluster $\alpha \notin A$. There are two cases here, namely, the case when α is not compatible with at least one of $C(T_1)$ and $C(T_2)$ and the case when α is compatible with both $C(T_1)$ and $C(T_2)$.

Now, when α is not compatible with at least one of $C(T_1)$ and $C(T_2)$, using Theorem 4.1, we observe that $\alpha \notin C(T_{olc})$.

For the second case, pick those smallest clusters $\alpha_1 \in C(T_1)$ and $\alpha_2 \in C(T_2)$ such that $\alpha \subseteq \alpha_1$ and $\alpha \subseteq \alpha_2$. (Note that the nodes v and u defining the clusters α_1 and α_2, respectively, are the LCAs in T_1 and T_2, respectively, of the species in α.) Since α_1 and α_2 are the smallest clusters in T_1 and T_2, respectively, containing α and since α is compatible with both $C(T_1)$ and $C(T_2)$, this implies that α is the union of clusters of at least two children of v and also the union of clusters of at least two children of u. Moreover, $\exists a, b \in \alpha$ such that $v = lca_{T_1}(a, b)$ and $u = lca_{T_2}(a, b)$. Furthermore, $\exists \beta \subseteq S, \beta \neq \emptyset$, such that $\alpha_1 \cap \alpha_2 = \alpha \cup \beta$. Thus we can pick a $c \in \beta$ and we have that $T_1|\{a, b, c\} = T_2|\{a, b, c\} = (a, b, c)$. But the topology given by having $\alpha \in C(T_{olc})$ is $((a, b), c)$. Thus $\alpha \notin C(T_{olc})$. □

4.1.2. Construction phase. Since the OLC rule is conservative, if the tree T_{olc} exists, then $C(T_{olc}) \cup C(T_1)$ is a compatible set of clusters, and hence there exists a tree T^* satisfying $C(T^*) = C(T_1) \cup C(T_{olc})$. If we can construct T^* by refining T_1, we can then reduce T^* by contracting all the unnecessary edges and thus obtain T_{olc}. This is the approach we will take.

Note that this approach breaks the construction into two stages: *refinement* and *contraction*.

DEFINITION 4.5. *We say that a tree T_1 is a refinement of tree T_2 if T_2 can be obtained from T_1 by a sequence of edge contractions.*

Refining T_1. The main objective is to refine T_1 so as to include all the clusters from T_{olc}. Before we explain how we do this precisely, we will introduce some notation and lemmas from previous works which enable us to do this efficiently.

DEFINITION 4.6. *Let v be an arbitrary node in a tree T with children $u_1, \ldots, u_k$. A* representative set *of v is any set $\{x_1, x_2, \ldots, x_k\}$ such that $x_i \in \alpha_{u_i}$. We denote by $rep(v)$ one such representative set.*

LEMMA 4.1. *If the OLC tree T_{olc} of trees T_1 and T_2 exists and $v \in T_1$, then $T_{olc}|rep(v)$ is isomorphic to $T_2|rep(v)$.*

Proof. The proof follows from the fact that $T_1|rep(v)$ is a star. □

DEFINITION 4.7. *Let v be a node in a tree T with children $u_1, u_2, \ldots, u_k$. Then $N(v)$ is the subtree induced by $\{v, u_1, u_2, \ldots, u_k\}$.*

We will do the refinement as follows. We will modify the tree T_1^*, where T_1^* is initialized to T_1. In a postorder fashion, for every $v \in V(T_1)$ with representative set $\{x_1, x_2, \ldots, x_k\}$, identify $v^* = lca_{T_1^*}(\alpha_v)$. It can be seen that v^* also has the same number of children as v (since the processing is done in a postorder fashion). Say these are $u_1, u_2, \ldots, u_k$. Replace the subtree $T(v^*)$, rooted at v^* in the following manner: we replace $N(v^*)$ by an isomorphic copy of $T_2|rep(v)$. Next, we replace x_i by the subtree of T_1^* rooted at u_i.

Let T^* be the tree that is produced after considering all the nodes in T_1.

THEOREM 4.3. *Let T_1, T_2 be given and suppose T_{olc} exists. Then the tree T^* that is produced from the algorithm described in the previous paragraph satisfies $C(T^*) = C(T_1) \cup C(T_{olc})$.*

Proof. Since $C(T_{olc}) \cup C(T_1)$ is compatible, all we need to show is that $T_{olc}|rep(v)$ cannot be a proper refinement of $T_2|rep(v)$. If it were, then for some $\{a, b, c\} \subseteq rep(v)$, $T_{olc}|\{a, b, c\}$ would be resolved while $T_2|\{a, b, c\}$ is unresolved. Since $\{a, b, c\} \subseteq rep(v)$, $T_1|\{a, b, c\}$ is also unresolved, forcing T_{olc} to be also unresolved. □

Note that we have reduced the problem of constructing T^* to the problem of discovering $T_2|rep(v)$ for each $v \in T_1$.

To have a linear time algorithm, however, we need to be able to compute $T_2|rep(v)$ quickly. We cite the following result from [18] which will be useful to us in this case.

LEMMA 4.2 (see [18]). *Given a left-to-right ordering of the leaves of a tree and the ability to determine the topology of any triple of leaves a, b, c in constant time, we can construct the tree in linear time.*

To use this lemma we need two things:

(1) we must be able to determine the topology of any triple in T_2 in $O(1)$ time and
(2) we must have for each node in T_1 an ordered representative set, where the ordering is consistent with the left-to-right ordering of the leaves in T_2.

To accomplish (1), we first preprocess T_2 for LCA queries. Then, to determine the topology for the triple a, b, c, we simply compare the LCAs of (a, b), (b, c), and (a, c). The second requirement is more challenging but can also be handled, as we now show.

Computing all ordered representative sets in $O(n)$ time.

- Initially all nodes in T_1 have empty labelings.
- For each $s \in S$, taken in the left-to-right ordering of the leaves in T_2, do the following steps:
  1. trace a path in T_1 from the leaf for s toward the root, until encountering either the root or a node which has already been labeled;
  2. append s to the ordered set for each such node in the path traced (including the first node encountered which has already been labeled).

Figure 2 shows an example of the computation just described.

Note that this computation takes $O(n)$ time since each node v is visited $O(\deg(v))$

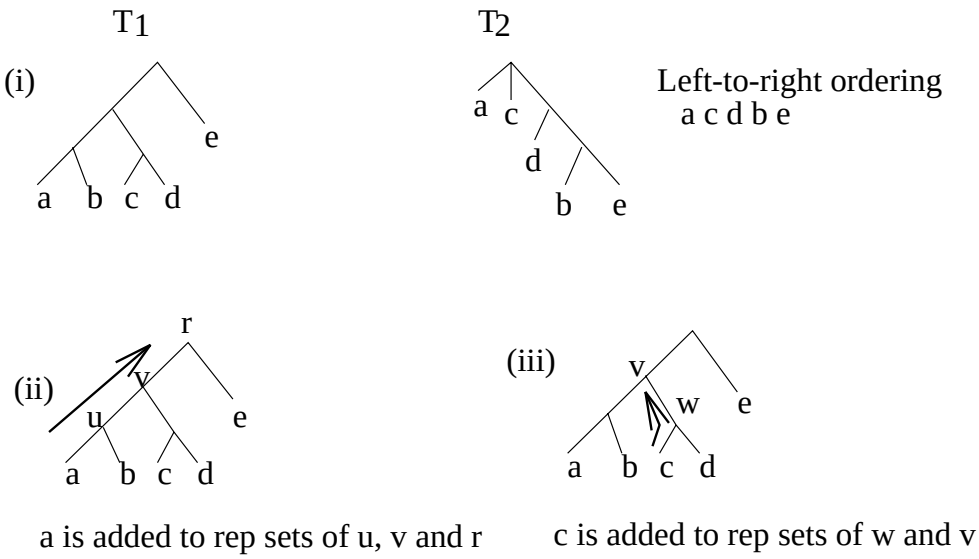

FIG. 2. *Example showing the computation of the representative sets of nodes in T_1 based on the left-to-right ordering of species in T_2.*

times and that the order produced is exactly as required. Thus, for each node $v \in V(T_1)$, we have defined a set of leaves such that each leaf is in a different subtree of v, every subtree of v is represented, and the order in which these leaves appear is the same as the left-to-right ordering in T_2.

We have thus proved Lemma 4.3.

LEMMA 4.3. *We can compute $T_2|rep(u)$ in $O(|rep(u)|)$ time.*

We therefore have the following theorem.

THEOREM 4.4. *Given T_1, T_2, then we can construct a tree T^* such that $C(T^*) = C(T_1) \cup C(T_{olc})$ whenever T_{olc} exists in $O(n)$ time.*

The rest of the task of constructing T_{olc} is in the contraction of unneeded edges.

Contracting T. Now that T^* satisfies $C(T^*) = C(T_1) \cup C(T_{olc})$, we can simply go through each edge in T^* and check if it needs to be kept or must be deleted. Note that edges that were added during the refinement phase are required and do not need to be checked. Therefore, we need only check the original tree edges. Let (u, v) be such an edge with $v = \text{parent}(u)$. From our representative sets for u and v we can easily choose three species a, b, c such that $lca(a, b) = u$ and $lca(b, c) = v$. If the topology of this triple in T_2 is resolved differently than $((a, b), c)$, then we know that edge (u, v) will have to be contracted; if on the other hand $T_2|\{a, b, c\}$ is either (a, b, c) or $((a, b), c)$ then (u, v) will have to be retained in any OLC tree.

OLC Construction Algorithm

Phase 0*: Preprocessing*

Make copies T_1' and T_2' of T_1 and T_2, respectively. For each node v in each tree T_i' $(i = 1, 2)$, compute ordered representative sets ordered by the left-to-right ordering in the other tree. Preprocess each tree T_i' to answer *lca* queries for leaves as well as internal nodes.

Phase I: *Refine* T_1'

Refine T_1' in a postorder fashion so that at the end $C(T_1') = C(T_1) \cup C(T_{olc})$ if T_{olc} exists.

Phase II: *Contract* T_1'

Contract edges $e \in E(T_1')$ such that c_e, the cluster below e, lies in $C(T_1) - C(T_{olc})$.

We have thus shown the following theorem.

Theorem 4.5. *The algorithm stated above constructs the OLC of two trees* T_1 *and* T_2 *if the OLC exists.*

Analysis of Running Time

Phase 0*: Preprocessing*

In [20], Harel and Tarjan give an $O(n)$ time algorithm for preprocessing trees to answer LCA queries in constant time. We have already shown that computing the ordered representative sets takes $O(n)$ time. Thus the preprocessing stage takes $O(n)$ time.

Phase I: *Refining* T_1'

This stage involves local refinements of T_1', and we have shown that the cost of refining around node v is $O(\deg(v))$. Summing over all nodes v we obtain $O(n)$ time.

Phase II: *Contracting edges*

This stage clearly takes only $O(n)$ time.

Theorem 4.6. *Construction of the optimistic local consensus tree can be done in linear time.*

4.1.3. Verification phase. We have identified a candidate optimistic local consensus tree. We now have to decide if this is really such a tree or that no such tree exists.

Lemma 4.4. *Let* T *be a tree on a leaf set* S*. Let* T^* *be obtained from* T *through a sequence of refinements followed by a sequence of edge contractions. Then there exists a function* $f : V(T) \to V(T^*)$ *such that for all* $v \in V(T)$*, there is a subset* S_v *of the children of* $f(v)$ *in* $V(T^*)$ *such that* $\alpha_v = \cup_{v' \in S_v} \alpha_{v'}$.

Proof. We define $f(v) = lca_{T^*}(\alpha_v)$. Clearly, $C(T^*) \cup C(T)$ is a compatible set of clusters. Therefore, there is a subset S_v of the children of $f(v)$ such that $\cup_{v' \in S_v} \alpha_{v'} = \alpha_v$. ∎

We take a slight detour and examine the verification of the OLC when the two input trees are both binary. In this case no triple will be unresolved.

Definition 4.8. *A* caterpillar *is a rooted binary tree with only one pair of sibling leaves.*

Given a leaf labeled caterpillar T' with root r and height h, there is a natural ordering induced by T' on its leaves. Let $g : S \to \{1, 2, \ldots, h\}$ be a function where $g(s)$ is the distance of s from r.

Then the species in S can be ordered in the increasing order as $a_1, a_2, \ldots, a_n$, where $a_i \in S$ such that $g(a_1) < g(a_2) \cdots < g(a_{n-1}) \leq g(a_n)$. (Note that the pair of sibling leaves have been arbitrarily ordered.)

DEFINITION 4.9. *Two caterpillars X and Y on the same leaf set are said to be* oppositely oriented *iff for all k, the k smallest elements of X are contained among the $k+1$ largest elements of Y and vice versa. See Figure* 3.

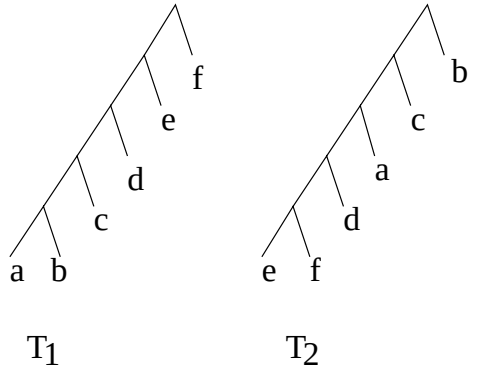

FIG. 3. *Example of oppositely oriented caterpillars.*

PROPOSITION 4.2. *Let T_1 and T_2 be two rooted binary trees on the same leaf set whose OLC is a star. If a, b is a sibling pair of leaves in T_1, then the LCA of a and b in T_2 must be the root of T_2.*

Proof. Suppose Proposition 4.2 is not true. Then there is a species c such that the LCA of (a, c) is above the LCA of (a, b) in T_2. Then $T_1|\{a, b, c\} = T_2|\{a, b, c\}$ and hence the OLC of T_1 and T_2 cannot be a star. □

LEMMA 4.5. *Suppose T_1 and T_2 are binary trees on the same leaf set and suppose that they each have at least five leaves. If their OLC tree is a star, then T_1 and T_2 must be caterpillars.*

Proof. Suppose for contradiction that T_1 is not a caterpillar. Then it has two pairs of sibling leaves (a, b) and (c, d). By the previous proposition each of these pairs must have the root as their LCA in T_2. Thus without loss of generality, a and c lie in the left subtree of the root of T_2, and b and d lie in the right subtree of the root of T_2.

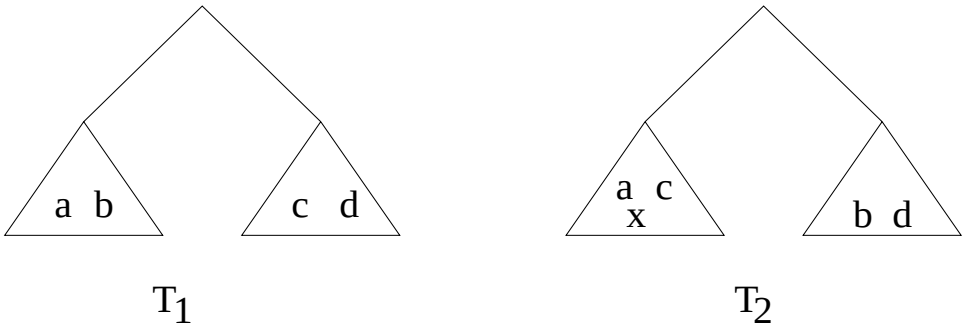

FIG. 4. *Topologies of T_1 and T_2 with respect to a, b, c, d, x.*

Let x be any other species besides a, b, c, and d (see Figure 4). Suppose without loss of generality that x lies in the left subtree of the root of T_2. We will consider the following two triples: x, a, d and x, c, b. In T_2 the topology of these triples will be $((x, a), d)$ and $((x, c), b)$, respectively.

We will show that T_1 agrees on at least one of these triples. There are two cases. If x lies in the left subtree of the root of T_1, then the topology of the triple x, a, d in T_1 is clearly $((x, a), d)$ and if x lies in the right subtree of the root of T_1, then the topology of the triple x, c, b in T_1 is $((x, c), b)$. Thus in either case there is a triple in T_1 which agrees with a triple in T_2, and the OLC cannot be a star. □

LEMMA 4.6. *Let T_1 and T_2 be two caterpillars on the same leaf set. Then the OLC of T_1 and T_2 is a star iff T_1 and T_2 are oppositely oriented caterpillars.*

Proof. Suppose the two caterpillars are oppositely oriented, i.e., they satisfy the two intersection conditions. Let x, y, z be any three leaves and let their indices in the ordering of the leaves of T_1 be $i < j < k$, respectively. Then the topology of x, y, and z in T_1 is $(x, (y, z))$. Looking at the $n - j$ smallest elements in T_2, this set must contain y or z but cannot contain x. Consequently, the topology of the triple in T_2 is not $(x, (y, z))$ and the star is a valid OLC.

Conversely, suppose that the two caterpillars do not satisfy the intersection conditions. Without loss of generality, suppose that there exists at least one k such that the k smallest elements of T_2 are not contained within the $k + 1$ largest elements of T_1. Pick the smallest such k. Say x is the leaf in T_2 with rank k and x does not belong to the set of $k+1$ largest elements of T_1. From the pigeonhole principle, there will exist at least two leaves of T_2 which have ranks greater than k but which are contained in the set of $k + 1$ largest elements of T_1. Suppose the two leaves are y and z. Then $T_1|\{x, y, z\} = T_2|\{x, y, z\} = (x, (y, z))$. This implies that the OLC cannot be a star. □

COROLLARY 4.1. *The OLC for two binary trees can be verified to be a star in linear time.*

Now we return to the general case of verifying the OLC of two trees.

LEMMA 4.7. *Suppose T is the OLC of T_1 and T_2 (on a leaf set S containing at least five species). Then T is a star iff either one of the following holds:*

1. *both T_1 and T_2 are oppositely oriented caterpillars or*
2. *both T_1 and T_2 are stars.*

Proof. The "if" direction is easy to see. We now assume that the OLC, T, is a star. If T_1 contains a triple a, b, c that is unresolved, T_2 must also be unresolved on a, b, c. Conversely whenever T_1 is resolved on a, b, c, T_2 must be (differently) resolved on a, b, c. Thus either both T_1 and T_2 are binary or both are not.

In the case that both T_1 and T_2 are binary, we appeal to the proofs of Lemmas 4.5 and 4.6 to argue that T_1 and T_2 must be oppositely oriented caterpillars.

If T_1 and T_2 are not binary, we will show that for any node v in T_1 with children $\{u_1, \ldots, u_k\}$, $k \geq 3$, there is a node v' in T_2 with children $\{u'_1, \ldots u'_k\}$ such that $\alpha_{u_i} = \alpha_{u'_i}$. Pick any three species a, b, c such that a, b, c is unresolved in T_1 and let $v = lca_{T_1}(a, b, c)$. Then a, b, c must be unresolved in T_2. Let $v' = lca_{T_2}(a, b, c)$. We claim that $\alpha_v = \alpha_{v'}$. To see why, suppose $\alpha_v \neq \alpha_{v'}$ and suppose $x \in \alpha_v, x \notin \alpha_{v'}$ with x being in the same subtree under v as a. Then $T_1|\{b, c, x\} = (b, c, x)$, whereas $T_2|\{b, c, x\} = ((b, c), x)$. This contradicts the assumption that T is a star. Thus $\alpha_v = \alpha_{v'}$.

Next, note that if x and y are under the same child of v in T_1 but under different children of v' in T_2, then there exists a z such that x, y, z is resolved in T_1 but unresolved in T_2. This would contradict the fact the T is a star. This establishes the claim.

This implies that if there is a nonbinary node v that is not the root of T_1, we can find two species a, b ($a \leq v, b \leq v$) and a species c, $c \not\leq v$ such $T_1|\{a, b, c\} = T_2|\{a, b, c\}$.

Thus the root must have three or more children in this case. But this means that if any cluster defined by a child of the root contains two or more species, then there is a triple on which T_1 and T_2 agree. Thus T_1 and T_2 must be stars. □

The verification proceeds as follows:

Phase 0

Suppose the tree constructed by refining T_1 and then contracting the edges in the resulting tree is T. We will do the same modification on T_2, i.e., refine T_2 using the information from T_1 and then contract the edges in the resulting tree as before. Call this tree $T^{'}$. Clearly, if T is not isomorphic to $T^{'}$, we can terminate and output that the OLC does not exist. This is because we know that a compatible set of clusters defines a unique tree and we know that the OLC, if it exists, is uniquely characterized.

Phase 1

If Phase 0 is successful, we then verify further. We compute an ordered representative set for every node w in $V(T)$. For each node w in T, do the following steps.

1. Check if the homeomorphic subtrees of T_1 and T_2 induced by $rep(w)$ are both stars or they are both oppositely oriented caterpillars. If they are neither of these, then terminate and output that the OLC does not exist.
2. Identify the parent of w, say w^*. Look at $rep(w^*)$ excluding the representative element which is below w. Call this set A. Identify the LCAs of $rep(w)$ in T_1 and T_2. Check if there is a species that belongs to A which lies below the LCA of $rep(w)$ in both T_1 or T_2. If so, terminate and output that the OLC does not exist.

Implementation of step 1 *of Phase* 1. Using the left-to-right ordering of the species in T_1, compute the ordered representative set rep at each node in T as shown in the previous section. For any $u \in V(T)$, to be able to quickly compute the homeomorphic subtree of T_2 induced by the species in $rep(u)$, we need to know the ordering of theses species as they appear in the left-to-right ordering of T_2. We associate with each u, a new rep set, $rep^*(u)$, which is the rearranged version of the species in $rep(u)$ according to their ordering in T_2. We define a function, $limit : S \rightarrow V(T)$, which specifies for each $s \in S$ the node $v \in V(T)$ closest to the root of T such that $s \in rep(v)$. The function $limit$ together with the left-to-right ordering of the species in T_2 help in filling the rep^* sets, since s will belong to the rep^* sets of all nodes in the path from s to $limit(s)$. We first show how to compute $limit(s) \forall s \in S$ using algorithm $LIMIT$ and then we show how the rep^* sets are filled.

Initialization:
$limit(s) = +\infty \forall s \in S$.

PROCEDURE LIMIT
For each $v \in V(T)$ visited in a top-down traversal of T,
do {
  Identify $rep(v)$
  For each $s \in rep(v)$ such that $limit(s) = +\infty$
    set $limit(s) = v$
}enddo

Once $limit(s)$ has been identified for all $s \in S$, we proceed to compute $rep^*(u) \forall u \in V(T)$ as follows. Look at the left-to-right ordering of the species in T_2. Now, for each species s in the left-to-right order, we trace a path in T from the leaf for s toward

the root of T and add s to the rep^* set of each node encountered in this path. We terminate when we reach $limit(s)$.

Note that this process of identifying rep and rep^* has to be done only once.

Analysis of running time. The isomorphism test in *Phase* 0 can be performed in $O(n)$ using a simple modification of the tree-isomorphism testing algorithm in [2].

There is an $O(n)$ cost for preprocessing of T_1 and T_2 to answer LCA queries in *Phase* 1.

Our implementation of *step* 1 of *Phase* 1 involves a one-time $O(n)$ cost in preprocessing to identify rep and rep^* for each node in T. Then each time *step* 1 is called on a node $w \in V(T)$, an additional time of $O(\deg(rep(w)))$ is taken.

Exploiting that fact that T_1 and T_2 have been preprocessed to answer LCA queries, it can be seen that each *step* 2 of *Phase* 1 takes $O(\deg(w) + \deg(w^*))$.

Thus the total time taken in the verification phase is $O(n)$.

Correctness of our verification procedure. See Theorem 4.7.

THEOREM 4.7. *If T passes the above tests, then T is the OLC of T_1 and T_2.*

Proof. We need only show that T handles every triple properly. Each of the following cases is handled assuming T has passed the isomorphism test.

Case 1. If T passes the isomorphism test with T', then any triple a, b, c such that the two trees resolve a, b, c differently will be unresolved in T. This follows since T is created by refining and then contracting both T_1 and T_2, and these actions cannot take a resolved triple into a different resolution.

Case 2. This involves a triple a, b, c having the same topology $((a, b), c)$ in both T_1 and T_2. We claim that the first step of *Phase* 1 will pass only if the topology of this triple is $((a, b), c)$. To see why, suppose a, b, c is unresolved in T. (a, b, c cannot be resolved as $(a, (b, c))$ or $((a, c), b)$ in T.) Look at the nodes u and v, which are the LCAs of a, b in T_1 and T_2, respectively. The node w in T, which is the $lca(a, b, c)$, is also $lca(a, b)$ (since a, b, c is unresolved). We infer that $f(u) = w$, where f is the function as defined in Lemma 4.4. This is because any node above w will contain the species c and any node below w will not contain either a or b. By a similar argument, $f(v) = w$. Now, when we look at $rep(w)$ and compute the homeomorphic subtrees of T_1 and T_2 induced by $rep(w)$, in both of these induced trees, there will exist three species x, y, z such that x, y are both below u (and v) in T_1 (and T_2) and z is not in the character defined by u (and v). Thus in both the induced trees, the triple x, y, z will have the same topology $((x, y), z)$. That is, these induced trees will neither be both stars nor both oppositely oriented caterpillars. Thus the verification process will terminate and output that the OLC does not exist.

Case 3. This involves a triple a, b, c which is resolved as $((a, b), c)$ in one tree and unresolved in the other. The proof of this case essentially follows the lines of the proof of Case 2.

Case 4. This involves a triple a, b, c which is unresolved in both trees. We claim that the second step of *Phase* 1 will pass only if this triple is unresolved in T. To see why, suppose a, b, c is resolved as $((a, b), c)$ in T. Let $lca_T(a, b, c) = x$ and let $lca_T(a, b) = y$ and also suppose without loss of generality that x is the parent of y. Let y_1 be the child of y such that $a \in \alpha_{y_1}$ and let y_2 be the child of y such that $b \in \alpha_{y_2}$. Let $z \neq y$ be the child of x such that $c \in \alpha_z$.

Let $u = lca_{T_1}(a, b, c)$ and $v = lca_{T_2}(a, b, c)$.

We will look at functions f_1 and f_2 defined by Lemma 4.4 from $V(T)$ to $V(T_1)$ and $V(T_2)$, respectively. Clearly $f_1(y) = u$ and $f_2(y) = v$. Note that the cluster defined by any child of u can have a nonempty intersection with at most one of α_{y_1}

and α_{y_2}. This is similar for v. Thus any representatives chosen from α_{y_1} and α_{y_2}, respectively, have their LCA at u in T_1 and at v in T_2. However, $f_1(z) \leq_{T_1} u$ and $f_2(z) \leq_{T_2} v$. Thus any representative chosen from α_z will lie below u and v in T_1 and T_2, respectively, causing us to conclude that the OLC does not exist. □

4.2. PLC. Recall the definition of the PLC tree: *Let T_1 and T_2 be two rooted trees on the same leaf set S. A rooted tree T is called the* PLC *of T_1 and T_2 iff for each triple a, b, c, $T|\{a, b, c\} = ((a, b), c)$ iff $T_1|\{a, b, c\} = T_2|\{a, b, c\} = ((a, b), c)$.*

Just like the OLC, the PLC tree need not always exist either.

4.2.1. Characterization. The following theorem characterizes the PLC tree of two trees T_1 and T_2.

THEOREM 4.8. *Let T_1 and T_2 be two trees on the same leaf set S. If the PLC tree T_{plc} of T_1 and T_2 exists, then it is identically equal to T, where $C(T) = C(T_1) \cap C(T_2)$.*

Proof. Pick any cluster $\alpha \in C(T)$. Since α belongs to both the trees, if we look at any triple x, y, z with $x, y \in \alpha$ and $z \notin \alpha$, then this triple will have to be resolved as $((x, y), z)$. Thus $\alpha \in C(T_{plc})$.

Conversely, pick any cluster $\alpha \notin C(T)$. We have two subcases here.

1. α is not compatible with at least one of $C(T_1)$ or $C(T_2)$. In this case, from Theorem 4.1, $\alpha \notin C(T_{plc})$.
2. α is compatible with both $C(T_1)$ and $C(T_2)$. In this case, pick those nodes from T_1 and T_2 that define the smallest clusters containing α. We can pick a triple a, b, c such that $a \in \alpha, b \in \alpha, c \notin \alpha$ and this triple is unresolved in either T_1 or T_2. Thus $\alpha \notin C(T_{plc})$. □

4.3. Construction phase. By Theorem 4.8, the PLC tree, if it exists, is identically the strict consensus tree. Thus to construct the PLC tree, it suffices to use the $O(n)$ algorithm in [9] for the strict consensus tree.

4.3.1. Verification phase. Let T_1 and T_2 be the input trees, and let T be the strict consensus tree constructed using the algorithm in [9]. We want to be able to verify whether T is actually the PLC in the case that T is a star. If T_1 or T_2 is already a star then there is nothing to verify since T is the true PLC. So assume that this is not the case.

There are two cases which we will consider. The first is when either of T_1 or T_2 (say T_1) has at least two children of the root which are not leaves. The second case is when both T_1 and T_2 have exactly one child of the root which is not a leaf. Having made observations about these cases, we can apply a divide and conquer strategy as seen by the following lemma.

LEMMA 4.8. *Let T_1 and T_2 be rooted trees on the same leaf set and let α be a cluster in their intersection. Let T be the strict consensus tree of T_1 and T_2. Let e_1, e_2, and e be the edges in T_1, T_2, and T respectively, that are above the respective internal nodes which define the cluster α. Let a be a species in α. Then T is a PLC for T_1 and T_2 iff*

(1) *the subtree below e is a PLC for the subtrees below e_1 and e_2, and*
(2) *upon replacing the subtrees below e, e_1, and e_2 by a in T, T_1, and T_2, respectively, T is a PLC for T_1 and T_2.*

Proof. Clearly, if T is the PLC tree for T_1 and T_2 then conditions (1) and (2) will hold. Conversely, if (1) and (2) hold, but T is not the PLC tree for T_1 and T_2, then there is some triple a, b, c such that T incorrectly handles this triple. If all of a, b, c are below e then by condition (1), T handles a, b, c correctly. Similarly if at least two are above e, then by condition (2), T handles this triple correctly. It remains to show

that T handles all triples where exactly two of a, b, c are below and one is above the edge e. But then, since the cluster $\alpha \in C(T_1) \cap C(T_2) = C(T)$, in each of T_1, T_2, and T, we have $((a, b)c)$, so that T handles this triple properly. Thus T is a PLC for T_1 and T_2. □

Thus the verification proceeds by traversing T in a postorder fashion and at the end of each successful verification step replacing the subtree by a single element belonging to the cluster defined by the root of the subtree. We now discuss the details of each verification step.

LEMMA 4.9. *Suppose T_1 and T_2 are two trees on the same leaf set S with T_1 having at least two children of the root which are not leaves. Let $\alpha_1, \ldots, \alpha_l$ be the maximal clusters of T_1 and $\beta_1, \ldots, \beta_m$ be the maximal clusters of T_2. Then T, their PLC, is a star iff $\forall i, j\ |\alpha_i \cap \beta_j| \leq 1$.*

Proof. Suppose $\forall i, j\ |\alpha_i \cap \beta_j| \leq 1$. This means that $\forall x, y$, if $lca(x, y)$ in T_1 is below the root, then in T_2, $lca(x, y)$ is the root. Thus for any triple x, y, z, their topologies in T_1 and T_2 do not agree. Thus T is a star.

Suppose $\exists i, j\ |\alpha_i \cap \beta_j| > 1$. Thus α_i is defined by a node which is not a leaf. Look at an α_k, $k \neq i$, such that the node in T_1 defining α_k is not a leaf node. There are two cases to handle here. Either at least one species in α_k is not in β_j or all species in α_k are in β_j (i.e., $\alpha_k \subset \beta_j$).

In the former case, pick that species z that is in α_k but not in β_j. Also pick those two species x, y that are in $\alpha_i \cap \beta_j$. Both T_1 and T_2 agree on the triple x, y, z; namely this triple has topology $((x, y), z)$ in both the trees. Thus T cannot be a star.

In the latter case, since we know that $\beta_j \neq S$, we can pick two species x, y from α_k and another species z from $S - \beta_j$. In both T_1 and T_2, the topology of this triple is $((x, y), z)$. Thus T cannot be a star. □

Since each species belongs to at most one of these maximal clusters in each tree, this test can be done in linear time.

The following lemma handles the case when both T_1 and T_2 have exactly one child of the root which is not a leaf.

LEMMA 4.10. *Suppose T_1 and T_2 are two trees on the same leaf set S and T and their PLC is a star. Suppose both T_1 and T_2 have exactly one child of the root each which is not a leaf. Let $s_1, \ldots, s_k$ be leaves in T_1 which are children of the root. Let v be the LCA in T_2 of $s_1, \ldots, s_k$. Then every child of v contains at most one species $x \in S - \{s_1, \ldots, s_k\}$. Moreover, for any pair of species $x, y \in S - \{s_1, \ldots, s_k\}$, the LCA of x and y in T_2 lies on the path from v to the root.*

Proof. Suppose $\exists$ a child of v which contains at least two species from $S - \{s_1, \ldots, s_k\}$. Then by picking x, y such that they both lie under this child if v in T_2 and picking an s_i out of $s_1, \ldots s_k$ that lies under a different child of v, we find that both trees have the same topology for the triple x, y, s_i. Thus T cannot be a star. Furthermore, if $\exists x, y \in S - \{s_1, \ldots, s_k\}$ such that $lca(x, y)$ in T_2 does not lie on the path from v to the root, then the triple x, y, s_1 would have identical topologies in both trees and T wouldn't be a star. □

DEFINITION 4.10. *A rooted tree T is a* millipede *if the set of internal nodes of T defines a single path from the root to a leaf. See Figure* 5.

Let $S_1 = S - \{s_1, s_2, \ldots, s_k\}$. We have that $T_2|S_1$ is a millipede (say, T_2^*).

Let $u_1, \ldots, u_l$ be the children of the root in T_2^*, which are leaves. Look at $T_1|S_1$ (say, T_1^*). Either, T_1^* has one nonleaf child or it has at least two nonleaf children. In the former case, we can apply the previous lemma and infer that $T_1^*|(S_1 - \{u_1, \ldots, u_l\})$ will be a millipede. In the later case, we can apply Lemma 4.10 to check if the PLC is a star.

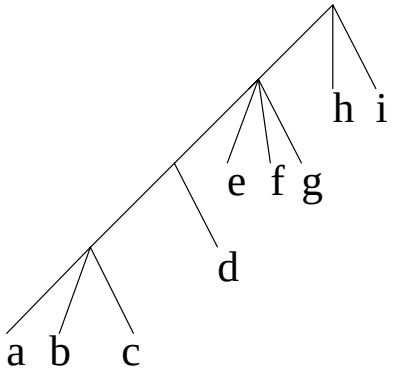

FIG. 5. *An example of a millipede.*

In the following subsection we will show how to verify if T is a star when both the input trees are millipedes.

4.3.2. Verification when both the input trees are millipedes. The proof of the following lemma is straightforward.

LEMMA 4.11. *Suppose T_1 and T_2 are two millipedes on the same leaf set S. Then their PLC T is a star iff there exists no triple such that both trees have the same resolved topologies on the triple.*

We now describe a linear time algorithm for verifying that T_1 and T_2 have no triple on which they have the same topology.

We define an ordering on the species in T_1 using the function $f : S \to \{1, \ldots, h\}$, where $f(s) =$ distance of s from the root of T_1 and h is the height of T_1.

In T_2, we can write S as the union of all the sets in the sequence $S_1, S_2, \ldots, S_k$, where k is the height of T_2 and each S_i contains exactly those species which are at a distance i from the root of T_2. Now, in each S_i replace each species s in this set with $f(s)$. Call this multiset of integers M_i. We thus get a sequence $M_1, M_2, \ldots, M_k$ of multisets.

DEFINITION 4.11. *We will say a triple of integers p, q, r is* special *if*

- $p < q$, $p < r$;
- $p \in M_{j_1}, q \in M_{j_2}, r \in M_{j_3}$, *with* $1 \leq j_1 < j_2 \leq k$ *and* $1 \leq j_1 < j_3 \leq k$.

We observe that the PLC of T_1 and T_2 is a star iff no special triple p, q, and r exists.

The following *CHECK_PLC* algorithm takes as input the sequence $M_1, M_2, \ldots, M_k$ and returns $FAIL$ if there exists a special triple of integers, and otherwise it returns $PASS$.

CHECK_PLC works by scanning the multiset M_i in the ith iteration. It makes use of three variables *global_min*, *local_min*, and *temp*. At the start of the ith iteration, *global_min* stores the smallest integer seen in the first $i - 1$ multisets. The variable *local_min* is used to store the smallest integer a such that $\exists b$ for which $a < b$ and $a \in M_j, b \in M_l$ with $1 \leq j < l < i$. (*local_min* is initialized to $+\infty$.) The variable *temp* is initialized to 0. As long as *temp* remains 0, *local_min* $= +\infty$. If *temp* is nonzero, then *local_min* stores a and *temp* stores some b for which the previously mentioned relationship between a and b holds. At the ith iteration, *CHECK_PLC* either returns $FAIL$ (if a special triple exists) or, if necessary, it modifies the variables *global_min*, *local_min*, and *temp* to hold their intended values for the first i multisets of the sequence.

The reasoning for storing these values at the start of the ith iteration is as follows. If $\exists p$ in some M_j, and $q, r \in M_i$ $(1 \leq j < i)$ such that p, q, r is a special triple, then *global_min* together with $q, r \in M_i$ are also a special triple since *global_min* $\leq p$. Similarly, if $\exists p$ in some M_j, $q \in M_l$, $r \in M_i$ $(1 \leq j < l < i)$, such that p, q, r is a special triple, then *local_min*, *temp*, and $r \in M_i$ are also a special triple.

We now describe *CHECK_PLC*.

Initialization:

$global\_min = Min(M_1)$
$local\_min = +\infty$
$temp = 0$.

The procedure outputs $FAIL$ (and terminates) if the PLC is not a star; it outputs $PASS$ otherwise.

PROCEDURE CHECK_PLC
For $2 \leq i \leq k$,
do {
If $temp = 0$, then *Step* 1, else *Step* 2.
Step 1
do {
Scan through M_i;
Identify $A = \{y | y \in M_i, global\_min < y\}$;
If $|A| \geq 2$, then output $FAIL$;
If $|A| = 1$, then set $temp = y$, where $y \in A$
$local\_min = global\_min$
$global\_min = Min\{global\_min, Min(M_i)\}$;
If $|A| = 0$, then set $global\_min = Min(M_i)$.
} enddo
Step 2
do {
Scan through M_i;
Identify $A = \{y | y \in M_i, global\_min < y\}$;
Identify $B = \{z | z \in M_i, local\_min < z\}$;
If either $|A| \geq 2$ or $|B| \geq 1$, then output $FAIL$;
Else
If $|A| = 1$ then
If $global\_min < Min(M_i)$, then set $local\_min = global\_min$
$temp = Min(M_i)$;
If $global\_min > Min(M_i)$, then set $local\_min = global\_min$
$temp = y$, where $y \in A$
$global\_min = Min(M_i)$;
If $|A| = 0$ then set $global\_min = Min(M_i)$.
} enddo
} enddo
Output $PASS$

Analysis of running time. *CHECK_PLC* runs in linear time since each M_i is scanned only a constant number of times.

THEOREM 4.9. *Algorithm* CHECK_PLC *is correct.*

Proof. By induction, observe that *Step* 1 is executed at the ith iteration if $\forall j, l, x$, where $1 \leq j < l < i$ and $x \in M_l$, $Min(M_j) \geq x$. It then follows that if *Step* 1 is executed at the ith iteration, then at the start of that iteration $temp = 0$,

$global\_min = Min(M_{i-1})$, and $local\_min = +\infty$. Thus, in this case $global\_min$ stores the smallest integer seen in the first $i-1$ multisets. Now, in the first i multisets, if any special triple p, q, r exists such that $p \in M_j$ $(j < i)$ and $q, r \in M_i$, then $CHECK\_PLC$ correctly outputs $FAIL$ since $global\_min \leq p$. Otherwise we have two cases, depending upon the value of A. If $|A| = 1$, then the variables $global\_min$, $temp$, and $local\_min$ are updated so that $global\_min$ holds the smallest value in the first i multisets. Also, $local\_min$ now correctly holds the smallest value a for which there exists a b (stored in $temp$) for which $a < b$ and $a \in M_j, b \in M_l$ with $1 \leq j < l < i$. In the other case $|A| = 0$, in which case $global\_min$ is updated to hold $Min(M_i)$ (which is the smallest value in the first i multisets).

Observe that once $temp$ is updated to store a nonzero value, it never stores a 0 again. Thus, once $temp$ is set to a nonzero value in iteration i', then from iteration $i'+1$ to iteration k, *Step* 2 is executed.

Assume that *Step* 2 is executed in some iteration i' and assume, inductively, that at the start of iteration i', $global\_min$ stores the smallest value in the first $i'-1$ multisets and $local\_min$ stores the smallest value a for which there exists a b (stored in $temp$) such that $a < b$ and $a \in M_j, b \in M_l$ with $1 \leq j < l < i'$. Then in iteration i', it can be easily seen that $CHECK\_PLC$ correctly outputs $FAIL$ if there exist a special triple p, q, r such that $p \in M_{i_1}, q \in M_{i_2}$ $(i_1 < i_2 < i')$, $r \in M_{i'}$ or $p \in M_{i_1}, q, r \in M_{i'}$ $(i_1 < i')$. Otherwise, for both the cases when $|A| = 1$ and $|A| = 0$, *Step* 2 ensures that after iteration i', $global\_min$ stores the smallest value in the first i' multisets and $local\_min$ stores the smallest value a for which there exists a b (stored in $temp$) such that $a < b$ and $a \in M_j, b \in M_l$ with $1 \leq j < l \leq i'$.

Using the above arguments, it can be seen that $CHECK\_PLC$ gives the correct output on any sequence of multisets. □

Thus we also have the following theorem.

THEOREM 4.10. *Given two millipedes T_1 and T_2, we can check if their PLC is a star in linear time.*

4.4. Summary. We have used three general techniques in constructing local consensus trees for these two total local consensus rules:

- we characterize the local consensus tree (that is, we define the set $C(T)$ of binary characters which encode the consensus tree T);
- we use the character encoding of the consensus tree if possible to construct the tree efficiently; and
- we verify that the constructed tree is the local consensus tree.

Some comments about the construction phase are in order. When working with conservative local consensus functions, assuming the local consensus tree T exists, it is possible to construct the local consensus tree T in two phases: a refinement phase in which one of the input trees T_i is refined to produce a tree T^* satisfying $C(T^*) = C(T_i) \cup C(T)$ and then edges are contracted in T^* to produce a tree T^{**} such that $C(T^{**}) = C(T)$.

5. Optimization problems.

5.1. Introduction. The local consensus rules we have seen so far are such that the output tree satisfying the constraints of a particular local consensus rule need not exist. Yet characterizing these rules and developing fast algorithms for them are important because if the consensus tree exists, then we can say something very concrete about it. The nonexistence of the consensus tree in all cases does motivate the need to look at the *optimization versions* of local consensus, where solutions

always exist. We will now describe some natural optimization problems for local consensus tree construction. In these problems, which we call *relaxed* versions, we will consider certain constraints to be absolutely required and let others be desirable but not required. Then we seek a tree meeting all the required constraints and as many of the desirable constraints as possible. We now define some obvious relaxed versions but note that many other versions are equally desirable.

Recall our discussion in section 3.3 regarding a profile being constant, compatible, and incompatible on a triple. The first optimization problem we consider is where we insist that all triples on which the profile is incompatible or is unresolved and constant are left unresolved, and then we seek to leave as resolved a maximal set of triples on which the profile is constant and resolved. This is relaxed version I (RV-I). The second problem is where we insist that all the triples, which the profile leaves as resolved and constant, be left resolved the same, and then we seek to leave a maximal set of the remaining triples as unresolved in the consensus tree. This is relaxed version II (RV-II). The third problem is where we insist that all triples on which the profile is incompatible or leaves unresolved and constant are left unresolved, and we seek to leave as resolved a maximal set of triples on which the profile is constant and resolved or is compatible. This is RV-III. In addition, RV-III also insists that all the resolved triples in the consensus tree be compatible with the profile. Finally, we look at an interesting rule LCR1, where we insist that all triples be left resolved on which the profile is constant and resolved or is compatible. This tries to capture the optimistic features of the OLC model. Unfortunately, the consensus tree need not always exist. We give a counterexample to show this.

5.2. Specific relaxed versions.

DEFINITION 5.1. *Let T_1 and T_2 be two rooted trees (not necessarily binary) on the same leaf set S. A rooted tree T is called an* RV-I *of T_1 and T_2 if whenever a triple a, b, c has differing topologies on T_1 and T_2, or both T_1 and T_2 leave a, b, c as unresolved, then that triple is unresolved in T and in addition T preserves the topology of a maximal set of triples which are resolved identically in T_1 and T_2.*

To prove the existence of an RV-I tree it is sufficient to show that there exists a tree where every triple on which T_1 and T_2 disagree is unresolved. The set of trees with this property can be partially ordered based on the set of triples (on which T_1 and T_2 agree) whose topology they preserve. Once this partial order is known to be nonempty, we have proved the existence of an RV-I since any maximal element in this partial order is such a consensus tree.

We note that if T has the star topology it leaves unresolved all triples on which T_1 and T_2 disagree. Hence the partial order is nonempty and the RV-I tree always exists. In section 5.3 we show that this tree is unique.

DEFINITION 5.2. *Let T_1 and T_2 be two rooted trees (not necessarily binary) on the same leaf set S. A rooted tree T is called an* RV-II *of T_1 and T_2 if T preserves the topology of all triples which are resolved identically in T_1 and T_2. In addition, T should leave unresolved a maximal set of triples on which T_1 and T_2 disagree or which are unresolved in both T_1 and T_2.*

Using an argument similar to the one used to prove the existence of an RV-I tree and noting that T_1 (or T_2) itself preserves the topology of all triples on which T_1 and T_2 agree, we conclude that the RV-II always exists. In section 5.4 we give an algorithm to construct the RV-II tree.

DEFINITION 5.3. *Let T_1 and T_2 be two rooted trees on the same leaf set S. Let T be a rooted tree on the same leaf set. Consider the following rules.*

Rule 1a. *If a triple a, b, c is resolved as $((a, b), c)$ in one tree and as $(a, (b, c))$ in the other, we require that it be unresolved.*

Rule 1b. *If a triple a, b, c is unresolved in both the trees, then we require that it be unresolved.*

Rule 2. *If a triple a, b, c is resolved as $((a, b), c)$ in one tree and is either resolved as $((a, b), c)$ or unresolved in the other tree, then we require it to be resolved as $((a, b), c)$.*

The tree T is called the relaxed version III (RV-III) *of T_1 and T_2 if*

1. *it always satisfies Rules* 1a *and* 1b *for triples;*
2. *it also satisfies Rule* 2 *for a maximal number of triples;*
3. *if a triple a, b, c is resolved as $((a, b), c)$ in T, then it is not resolved as $(a, (b, c))$ or $((a, c), b)$ in either T_1 or T_2.*

In section 5.5 we will show that an RV-III tree also always exists and is unique.

In the next subsections, we will look at the different relaxed versions in greater detail.

5.3. RV-I. In this subsection we will show that the RV-I of two rooted trees T_1 and T_2 is actually the strict consensus of these two trees.

THEOREM 5.1. *If T_1 and T_2 are two rooted trees, then their RV-*I *tree T always exists and is identically the strict consensus of T_1 and T_2.*

Proof. The existence of the RV-I tree T, was shown in section 5.2. Now we show that this tree is the strict consensus tree. Suppose there exists a triple a, b, c resolved differently in T_1 and T_2 as, say, $((a, b), c)$ and $(a, (b, c))$ (or (a, b, c)), respectively. Say the $lca_{T_1}(a, b) = u$ and $lca_{T_2}(b, c) = v$. Clearly, neither α_u nor α_v is in the strict consensus tree. Thus the strict consensus tree leaves unresolved any triple that has different topologies in T_1 and T_2.

Let T' be a tree in which for every triple a, b, c on which T_1 and T_2 differ, T' has an unresolved topology on this triple. Now suppose it is possible that T' contains a cluster that is not in $C(T_1) \cap C(T_2)$. Let α be this cluster and suppose without loss of generality that α is not a cluster of T_1. In T', for any pair of species $x, y \in \alpha$ and species $z \notin \alpha$ the topology has to be $((x, y), z)$. However, if this is also the case in T_1, then T_1 must also possess the cluster α contradicting our assumption. Thus there must exist a pair of species $x, y \in \alpha$ and a species $z \notin \alpha$ such that in T_1 their topology is not $((x, y), z)$. But this implies that T' cannot be an RV-I. Hence any candidate T' for an RV-I can only contain the clusters in the intersection of the cluster sets of T_1 and T_2.

If T' contains a proper subset of the clusters in the intersection of the sets of clusters of T_1 and T_2, then there exists a triple a, b, c on which T' has an unresolved topology while the strict consensus tree has a resolved topology that agrees with the topologies of T_1 and T_2. Hence the strict consensus of T_1 and T_2 is the RV-I tree of T_1 and T_2. □

As a consequence, the RV-I can be constructed in $O(n)$ time using the algorithm in [9], and there is no need to verify that the tree constructed is correct.

5.4. RV-II. In the RV-II problem we require that any triple on which the trees T_1 and T_2 agree must have its topology preserved in the consensus tree T. Further T should leave unresolved a maximal set of triples on which T_1 and T_2 disagree or both leave unresolved.

Previously we showed that the RV-II exists. We note that the RV-II tree is not unique. The construction of the RV-II can be accomplished by defining the set $A = \{((a, b), c) : T_1|\{a, b, c\} = T_2|\{a, b, c\} = ((a, b)c)\}$. This set of rooted triples can then be passed to the algorithm of Aho et al. [3], which computes a tree (if it exists)

having the required form on every triple in the set and also leaving a maximal set of additional triples outside that set unresolved. The algorithm in [3] takes $O(pn)$ time where $p = |A|$. Recall the proof of Theorem 3.1 for a description of the algorithm. Since in our case $p \in O(n^3)$, the use of the algorithm of [3] would result in a running time of $O(n^4)$. We will obtain a speedup to an $O(n^2)$ algorithm (which includes the verification) for the construction of the RV-II tree by using the fact that the tree necessarily exists.

5.4.1. An improved algorithm for RV-II. We will now describe an $O(n^2)$ time algorithm to construct an RV-II tree. We start by making a few observations about the RV-II tree T constructed by the algorithm of [3].

We will use α's to denote the clusters in T_1 and β's to denote the clusters in T_2. Suppose α and β are maximal clusters in T_1 and T_2, respectively, and suppose $\alpha \cup \beta \neq S$. Then we claim that $\alpha \cap \beta$ (if nonempty) will be a maximal cluster in T. This is because $\exists a \in S-(\alpha \cap \beta)$ such that $\forall x, y \in (\alpha \cap \beta)$, $T_1|\{x, y, a\} = T_2|\{x, y, a\} = ((x, y), a)$ and thus the elements of $(\alpha \cap \beta)$ all belong to one component of the graph which is constructed in the execution of the algorithm of [3]. Furthermore, $(\alpha \cap \beta)$ is exactly equal to one component of this graph since the algorithm never adds an edge between two nodes in the graph unless it is forced to and it can be seen that no element x in $(\alpha \cap \beta)$ is such that $\exists y, a \in S-(\alpha \cap \beta)$ with $T_1|\{x, y, a\} = T_2|\{x, y, a\} = ((x, y), a)$.

Thus, if $\alpha \cup \beta \neq S$, then $\alpha \cap \beta$ (if nonempty) is a maximal cluster in T. The case where $\alpha \cup \beta = S$, $\alpha \cap \beta \neq \emptyset$, can occur for at most one child of the root of T_1 and one child of the root of T_2 as the following lemma shows.

LEMMA 5.1. *Let T_1 and T_2 be two trees on the same leaf set S. Let $\alpha_1, \ldots, \alpha_k$ be the maximal clusters of T_1 and $\beta_1, \ldots, \beta_l$ be the maximal clusters of T_2. Then the case where $\alpha_i \cup \beta_j = S$, $\alpha_i \cap \beta_j \neq \emptyset$ can occur for at most one i and one j.*

Proof. Suppose not. Let $\alpha_i \cup \beta_j = S$, $\alpha_i \cap \beta_j \neq \emptyset$, $\alpha_{i^*} \cup \beta_{j^*} = S$, and $\alpha_{i^*} \cap \beta_{j^*} \neq \emptyset$, perforce with $i \neq i^*$ and $j \neq j^*$. Since $\alpha_i \cap \alpha_{i^*} = \emptyset$, we have that $\alpha_i \subseteq \beta_{j^*}$. But since $\alpha_i \cap \beta_j \neq \emptyset$, this implies that $\beta_j \cap \beta_{j^*} \neq \emptyset$. This is a contradiction since β_j and β_{j^*} are clusters defined by the children of the root and hence should be disjoint. □

Recall that the maximal clusters form a partition of the species set S (in each of T_1, T_2, and T). Also, from the above discussions we have that (i) $\alpha \cup \beta \neq S$ implies that $\alpha \cap \beta$ is a maximal cluster in T and (ii) there can be at most one case for which $\alpha \cup \beta = S$. These observations imply that in the case when $\alpha \cup \beta = S$, then $\alpha \cap \beta$ is the union of some maximal clusters of T.

With the above characterization a high-level description of the algorithm to construct T can be given as follows.

RV-II CONSTRUCTION ALGORITHM

1. For each pair of maximal clusters $\alpha \in C(T_1)$ and $\beta \in C(T_2)$ such that $\alpha \cap \beta \neq \emptyset$ and $\alpha \cup \beta \neq S$, recursively compute the tree on $\alpha \cap \beta$ and make its root a child of the root of T.
2. If there are maximal clusters α and β such that $\alpha \cup \beta = S$ but $\alpha \cap \beta \neq \emptyset$, compute the partition of $\alpha \cap \beta$; recursively compute the tree for each component of the partition and make the roots of these trees children of the root of T.

Computing the partition of $\alpha \cap \beta$ in step 2 is described together with the implementation details.

Implementation details and running time analysis. Note that this algorithm does not require an explicit verification of the constructed tree, since in fact we know that the tree exists and we are simply computing it by mimicking efficiently what the algorithm in [3] would create.

There are at most n recursive stages. We will show that each stage can be implemented in $O(n)$ time thereby proving the $O(n^2)$ bound.

To handle case 1 it is important not to waste time on empty intersections. So we consider each species in turn and label the intersection in which that this species lies. Thus we will identify at most n nonempty intersections. Let $\alpha \cap \beta$ be one such intersection. To recurse, we need to find homeomorphic subtrees of T_1 and T_2 that have $\alpha \cap \beta$ as the leaf set. We will show how to do this in time proportional to the number of leaves in $\alpha \cap \beta$.

Assume that T_1 and T_2 have been preprocessed for LCA queries. Also note that we know the left-to-right ordering of all leaves of T_1 as well as of T_2. Given the leaves in $\alpha \cap \beta$, their left-to-right ordering is also known and is the one induced by the overall left-to-right ordering. By Lemma 4.2 we can reconstruct the topology of the tree in linear time.

Thus case 1 can be handled in $O(n)$ time.

We now describe how to handle case 2 also in $O(n)$ time. We will construct a graph $G = (V, E)$ such that $V(G) = \alpha \cap \beta$. The edges will be added so that, finally, each component in G corresponds to a maximal cluster in the RV-II tree.

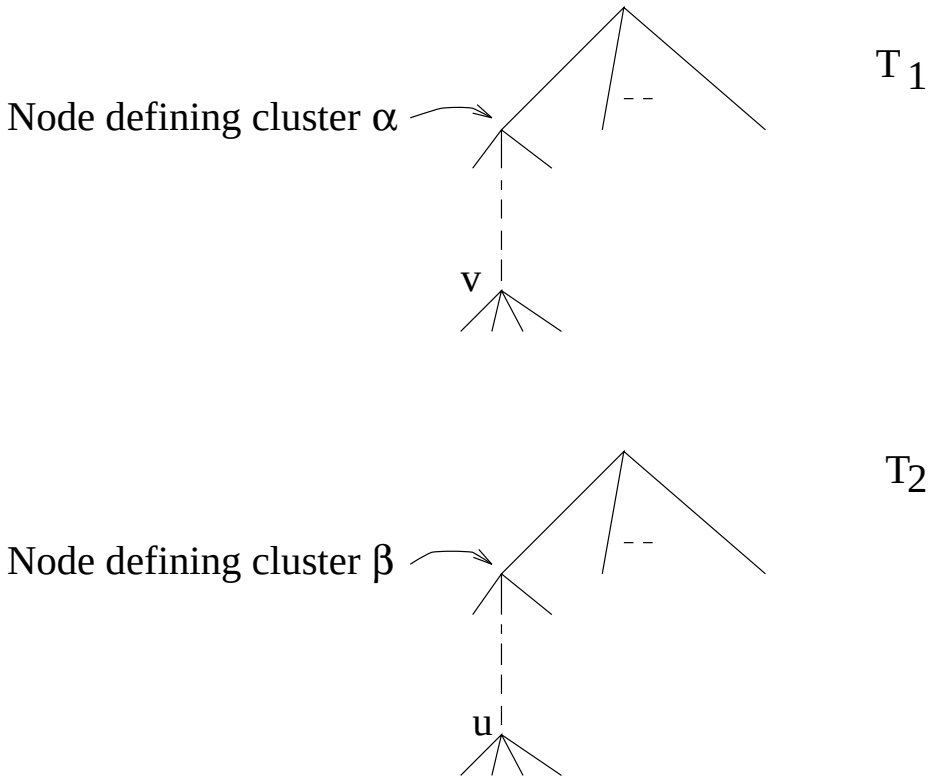

FIG. 6. *Figure showing nodes v and u.*

Identify the LCA, say, u, of the species in $S - \alpha$ in T_2 and similarly the LCA, say, v, of the species in $S - \beta$ in T_1. In T_2, u will be a descendent of the node defining β, and in T_1, v will be a descendent of the node defining α. See Figure 6. In T_1 let v_1 through v_p be the nodes in the path from the root to v, where $v_1 = root$ and $v_p = v$. Similarly, in T_2, let u_1 through u_q be the nodes in the path from the root to u, where

$u_1 = root$ and $u_q = u$. We will say that δ is a *special cluster* if for some v_i, $1 \le i \le p$ (or some u_j, $1 \le j \le q$), δ is a cluster defined by a child of v_i (or u_j) that is not on the path from the root to v (or u).

Let $\delta_1, \ldots, \delta_l$ be the special clusters in T_1 and let $\gamma_1, \ldots, \gamma_m$ be the special clusters in T_2. A pair of species $x, y \in (\alpha \cap \beta)$ will be in the same component of the graph G if $\exists z$ such that $T_1|\{x, y, z\} = T_2|\{x, y, z\} = ((x, y), z)$. There are two cases depending on whether $z \in (\alpha \cap \beta)$ or not. We will now describe how to handle these two cases: Cases 2a and 2b.

Case 2a [$z \notin (\alpha \cap \beta)$]. In this case, it suffices to look at all $\alpha \cap \gamma_i$ and $\beta \cap \delta_j$, and for each intersection put its elements in the same component of G. This is evident from the following lemma.

LEMMA 5.2. *Let α, β be maximal clusters of T_1 and T_2, respectively, such that $\alpha \cup \beta = S$ and let $x, y \in (\alpha \cap \beta)$. Then $\exists z \in S - (\alpha \cap \beta)$ such that $T_1|\{x, y, z\} = T_2|\{x, y, z\} = ((x, y), z)$ iff both x and y belong to some $\alpha \cap \gamma_i$ or $\beta \cap \delta_j$.*

Proof. Suppose $\exists z \in S - (\alpha \cap \beta)$ such that $T_1|\{x, y, z\} = T_2|\{x, y, z\} = ((x, y), z)$. Since $\alpha \cup \beta = S$, the only cases we have to consider are when z is in exactly one of α or β. So suppose $z \in \alpha, z \in S - \beta$ (the other case can be handled similarly). Then z belongs to a special cluster δ_i, which is defined by some child of the node v in T_1. (Recall that node v is the LCA of $S - \beta$ in T_1.) Since $T_1|\{x, y, z\} = ((x, y), z)$, we have that either both x, y belong to δ_i or neither belongs to δ_i. If both $x, y \in \delta_i$, then clearly $x, y \in (\beta \cap \delta_i)$. For the case when neither x nor y is in δ_i, we can conclude that both x, y are in some special cluster δ_j (since $T_1|\{x, y, z\} = ((x, y), z)$). Thus we have that $x, y \in (\beta \cap \delta_j)$.

Suppose x, y belong to some $\alpha \cap \gamma_i$ or $\beta \cap \delta_j$; specifically, say x, y belong to some $\beta \cap \delta_j$. There are two cases to handle. The first case is if the node v' defining the special cluster is not a child of the node v. In this case, we can pick a species $z \in S - \beta$ such that $T_1|\{x, y, z\} = T_2|\{x, y, z\} = ((x, y), z)$. The second case is when the node v' is a child of the node v. In this case, pick a species $z \in S - \beta$ from the special cluster which is defined by a node v'' (where $v' \ne v''$) and v'' is below v. We have that $T_1|\{x, y, z\} = T_2|\{x, y, z\} = ((x, y), z)$. Thus in both the cases we have that there exists such a z with $z \in S - (\alpha \cap \beta)$. □

Thus, for each i, connect all vertices in $\alpha \cap \gamma_i$ (in G) by a path and do the same for each j and the vertices in $\beta \cap \delta_j$. Note that this can be done in $O(n)$ by using the same idea as in Case 1.

Case 2b [$z \in (\alpha \cap \beta)$]. Note that we are only interested in identifying x, y such that $lca(x, y)$ in T_1 is a node that is on the path from the root of T_1 to the node v, and the $lca(x, y)$ in T_2 is a node that is on the path from the root of T_2 to the node u. To see why, if, say, $lca_{T_1}(x, y) \ne v_i \forall 1 \le i \le p$, then $\exists a \in S - \beta$ (i.e., $a \notin (\alpha \cap \beta)$) such that $T_1|\{x, y, a\} = T_2|\{x, y, a\} = ((x, y), a)$, and thus x and y will be in the same component after Case 2a is handled.

From the preceding discussion, it suffices to convert the trees T_1 and T_2, both defined on the leaf set $(\alpha \cap \beta)$, into millipedes T_1' and T_2', respectively. T_1' is obtained from T_1 by contracting all edges above internal nodes not in the set $\{v_1, v_2, \ldots, v_p\}$. T_2' is obtained from T_2 similarly. Thus, we have to solve the following problem now: we are given two millipedes T_1' and T_2' on the same leaf set $S' = (\alpha \cap \beta)$, where T_1' has internal nodes labeled v_1' (root of T_1') through v_p', and each v_i' has leaves corresponding to all the species in the special clusters of v_i in T_1; T_2' has internal nodes labeled u_1' (root of T_2') through u_q' and is defined similarly. Our aim is to construct a graph $G' = (V', E')$ where $V' = S'$ such that if $\exists x, y, z \in (\alpha \cap \beta)$ such that both T_1' and

T_2' resolve this triple as $((x, y), z)$ then x and y will be in the same component of G'. Once G' is known, we add the edges of G' to the edge set of G, and then the components in G will give the maximal clusters we seek.

We will show how G' can be constructed in $O(n)$ time. Consider a node v'_{i-1} in T_1' and let A be the set of leaves of v'_{i-1}. Let u'_{j-1} be the node in T_2' which is closest to u'_1 and is the parent of some species in A. Then, clearly, in G' all species in $(\alpha_{v'_i} \cap \beta_{u'_j})$ need to be in one component. For every v'_{i-1} $(2 \leq i \leq p)$, we will denote this intersection by the pair (v'_i, u'_j). Further, observe that if (v'_i, u'_j) and $(v'_{i'}, u'_{j'})$ are such that $v'_{i'}$ is not above v'_i and $u'_{j'}$ is not above u'_j, then $(\alpha_{v'_{i'}} \cap \beta_{u'_{j'}}) \subseteq (\alpha_{v'_i} \cap \beta_{u'_j})$.

Thus, when constructing the graph G', we need only look at all the intersections of the form (v'_i, u'_j), where for every pair of intersections $(v'_{i'}, u'_{j'})$ and (v'_i, u'_j), v'_i is closer to v'_1 than $v'_{i'}$ is, iff $u'_{j'}$ is closer to u'_1 than u'_j is.

Let $(v_1^*, u_1^*), (v_2^*, u_2^*), \ldots, (v_r^*, u_r^*)$ be the intersections we are interested in, where v_i^* is closer to v'_1 than v_{i+1}^* is $(1 \leq i \leq (r-1))$, and u_{j+1}^* is closer to u'_1 than u_j^* is $(1 \leq j \leq (r-1))$. Note that $v_1^* = v'_2$. This node and the given T_1' and T_2', uniquely determine these intersections.

In T_1', we define the *nearest parent* of a species x to be the first v_i^* to appear on the path from x to the root of T_1'. Similarly, we can define the nearest parent of a species in T_2'. The nearest parents of all the species can be computed in $O(n)$ by doing a simple traversal of T_1' and T_2'. Using the nearest parents of the species in T_1', we partition the species set into r sets $S_{v_1^*}, \ldots, S_{v_r^*}$ where $S_{v_i^*}$ contains all species which have nearest parent as v_i^*.

Observe that if any two intersections (v_i^*, u_i^*) and $(v_{i'}^*, u_{i'}^*)$ contain at least one species in common, then all the species in the two intersections need to be in the same component in G'. Inductively, if there are intersections $(v_i^*, u_i^*), \ldots, (v_j^*, u_j^*)$ such that the species in these intersections need to be in one component in G' and if there is an intersection (v_k^*, u_k^*) which has a species x in common with one of these intersections, then all the species in the intersection (v_k^*, u_k^*) need to be in the same component as the species in the intersections $(v_i^*, u_i^*), \ldots, (v_j^*, u_j^*)$. The algorithm *CONSTRUCT_G'* we present now keeps track of such an x using the variable *missing_link*, which is initialized to an $x \in (v_r^*, u_r^*)$ such that the nearest parent of x in T_2' (say u_j^*) is farthest from the root (as compared with the nearest parents of the other species in (v_r^*, u_r^*)). We will also use two additional variables: *np_missing_link* which stores u_j^* and *upper_limit* which stores v_j^*.

PROCEDURE CONSTRUCT_G'

For $i = r$ down to 1,
do{
Identify $y \in S_{v_i^*}$ such that the nearest parent of y in T_2' is
farthest away from u'_1 (i.e., root of T_2')
Let u_k^* be the nearest parent of y in T_2'; Set $z = v_k^*$
Connect all $x \in S_{v_i^*}$ to y
If *upper_limit* is not below v_i^*,
then
connect y to *missing_link*
else if (*upper_limit* is below v_i^*) or (*upper_limit* is below v_k^*)
then
set *missing_link* $= y$
np_missing_link $= u_k^*$
upper_limit $= z$
}enddo

Once we have constructed G', we can update G by setting $E(G) = E(G) \cup E(G')$. The components in G will be the maximal clusters of the RV-II. Finding the components takes $O(n)$. To recurse, we find the homeomorphic subtrees of T_1 and T_2 induced by the species in each of the maximal clusters. This can be done in $O(n)$ as previously described.

Thus the RV-II can be constructed in $O(n^2)$.

5.5. RV-III.

LEMMA 5.3. *The* RV-III *tree T of two trees T_1 and T_2 always exists and is unique. Further $C(T) = A$, where $A = \{\gamma | \gamma = \alpha \cap \beta, \alpha \in C(T_1), \beta \in C(T_2), \gamma$ compatible with $C(T_i), i = 1, 2\}$.*

Proof. We will first show that A as defined above is a compatible set. The uniqueness will then follow from the uniqueness of a set of compatible clusters [17].

Pick two clusters $\gamma_1 = \alpha_1 \cap \beta_1$ and $\gamma_2 = \alpha_2 \cap \beta_2$ such that $\gamma_i \in A; \alpha_1, \alpha_2 \in C(T_1); \beta_1, \beta_2 \in C(T_2)$. We will show that $\gamma_1 \cap \gamma_2 \in \{\emptyset, \gamma_1, \gamma_2\}$. Now, since γ_i is compatible with $C(T_1)$ and $C(T_2)$, we have $\gamma_1 \cap \alpha_2 \in \{\emptyset, \gamma_1, \alpha_2\}$. Also, we have $\gamma_1 \cap \beta_2 \in \{\emptyset, \gamma_1, \beta_2\}$. There are several cases to handle. The first case is when $\gamma_1 \subseteq \alpha_2, \gamma_1 \subseteq \beta_2$. In this case, $\gamma_1 \subseteq (\alpha_2 \cap \beta_2)$ or $\gamma_1 \cap \gamma_2 = \gamma_1$. The second case is when $\gamma_1 \supseteq \alpha_2, \gamma_1 \supseteq \beta_2$. In this case, $(\alpha_2 \cap \beta_2) \subseteq \gamma_1$ or $\gamma_1 \cap \gamma_2 = \gamma_2$. The third case is when $\gamma_1 \subseteq \alpha_2, \gamma_1 \supseteq \beta_2$. In this case, $(\alpha_2 \cap \beta_2) \subseteq \gamma_1$ and thus $\gamma_1 \cap \gamma_2 = \gamma_2$. Hence, A is a compatible set of clusters.

Now we will show that any tree T satisfying the RV-III rules will have its cluster encoding equal to A. From the third requirement for RV-III,[5] all the clusters in $C(T)$ are compatible with both $C(T_1)$ and $C(T_2)$. Now suppose we can pick a $\gamma \in C(T) - A$. This means that $\gamma \neq \alpha_i \cap \beta_j \forall \alpha_i \in C(T_1), \beta_j \in C(T_2)$. Let α_1 and β_1 be the minimal clusters in T_1 and T_2, respectively, containing γ. Clearly, $\alpha_1 \cap \beta_1 \supset \gamma$. Let u and v be the nodes in T_1 and T_2, respectively, which define the clusters α_1 and β_1. Since γ is compatible with $C(T_1)$ and $C(T_2)$, it follows that we can pick three species a, b, c such that $lca_{T_1}(a, b) = lca_{T_1}(a, c) = lca_{T_1}(b, c) = u$, $lca_{T_2}(a, b) = lca_{T_2}(a, c) = lca_{T_2}(b, c) = v$, and $a, b \in \gamma, c \in (\alpha_1 \cap \beta_1) - \gamma$. In both T_1 and T_2, the triple a, b, c is unresolved, but it is resolved as $((a, b), c)$ in T, thus contradicting the assumption that T' satifies the rules defined by RV-III. Thus we have that $C(T) \subseteq A$. Now suppose $C(T) \subset A$. Then it can be seen that we can pick a triple a, b, c which is resolved in T_1 and is either resolved the same in T_2 or is unresolved in T_2 but that a, b, c is unresolved in T. This contradicts the assumption that T satisfies the rules defined by RV-III since it does not satisfy the second (see definition of RV-III) for a maximal set of triples. Thus $C(T) = A$. ∎

LEMMA 5.4. *The RV-*III *tree T of two rooted trees can be computed in $O(n^3)$.*

Proof. We can compute $C(T)$ in $O(n^3)$ as follows. The set $X = \{\gamma | \gamma = \alpha \cap \beta, \alpha \in C(T_1), \beta \in C(T_2)\}$ can be computed in $O(n^3)$, since there are $O(n^2)$ pairs to look at and each $\alpha \cap \beta$ can be computed in $O(n)$. The set $Y = \{\gamma | \gamma \in X, \gamma$ compatible with $C(T_i)\}$ can be computed from X in $O(n^3)$, since each of the $O(n^2)$ clusters in X can be checked for compatibility with $C(T_i)$ in $O(n)$. Finally, T can be constructed from Y using the $O(n^2)$ algorithm mentioned in [17]. Thus the total time taken is $O(n^3)$. ∎

We now briefly discuss another local consensus rule that looks interesting but unfortunately does not always exist. We define $LCR1$ as a rule which requires that if

[5] If a triple a, b, c is resolved as $((a, b), c)$ in T, then it is not resolved as $(a, (b, c))$ or $((a, c), b)$ in either T_1 or T_2.

a triple a, b, c is resolved as $(a, (b, c))$ in one tree and is either resolved as $(a, (b, c))$ or unresolved in the second tree, then it is resolved as $(a, (b, c))$ in the consensus tree.

Although the above rule tries to capture the *optimistic* features of the input trees and at the same time is not a total local consensus rule, it is the case that the consensus tree defined by $LCR1$ need not exist. See Figure 7 for an example showing that $LCR2$ need not necessarily produce a tree. Figure 7(iii) shows the graph constructed by the algorithm in [3]. Since the graph is connected, it follows that the set of triple constraints does not define a tree.

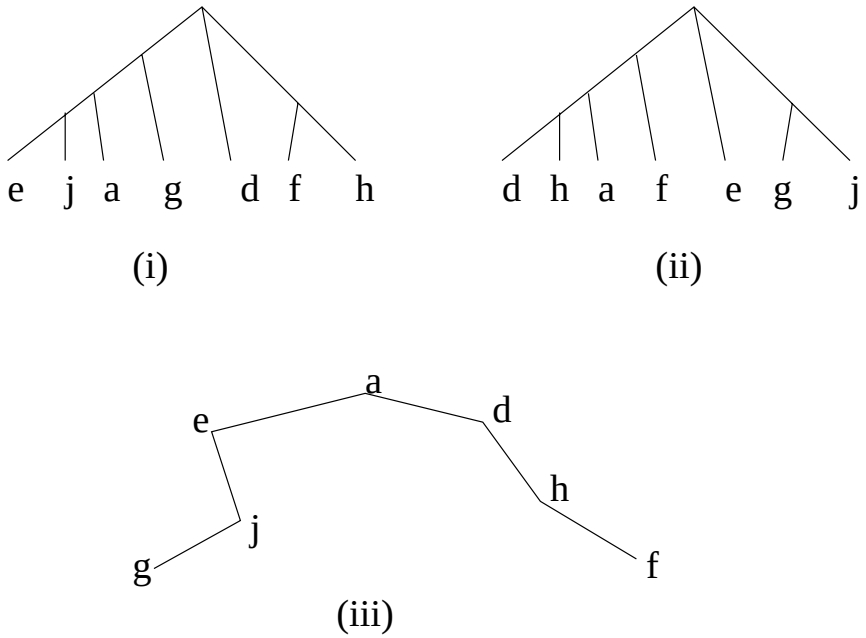

FIG. 7. *Example showing that consensus tree defined by $LCR1$ need not exist.*

6. Discussion and conclusions. Several approaches have been taken to handle the problem of resolving multiple solutions. One approach has been to find a maximum subset $S_0 \subseteq S$ inducing homeomorphic subtrees; this subtree is then called a *maximum agreement subtree* [19, 13, 24, 14]. The primary disadvantage of this approach is that it does not return an evolutionary tree on the entire species set.

The other approach which we take here requires that the resolution of the inconsistencies be represented in a single evolutionary tree for the entire species set. A classical problem in this area is the *tree compatibility problem* (also called the *cladistic character compatibility problem*) [10, 11, 12]. The tree compatibility problem says that the set $\mathcal{T}$ of trees is *compatible* if a tree T exists such that $C(T) = \cup_{T_i \in \mathcal{T}} C(T_i)$. Equivalently, if a tree T exists such that for every triple $A \subseteq S$, T resolves A iff $T|A = T_i|A$ for every $T_i \in \mathcal{T}$ which resolves A. This problem can be solved in linear time [17, 25]. The weakness of this approach is that in practice many data sets are incompatible, and it is therefore necessary to be able to handle the case where some pairs of trees resolve triples differently.

Some other approaches of this type are the *strict consensus* [4, 9] and the *median tree* [5] problems. These models are stated in terms of unrooted trees, so that instead of *clusters*, *characters* (i.e., bipartitions) on the species set are used to represent the

trees. Using the character encoding of the consensus tree as a measure of fitness to the input, the strict consensus seeks a tree with only those characters that appear in every tree in the input. The median tree, on the other hand, is defined by a metric $d(T_1, T_2)$ between rooted trees which is defined to be the cardinality of the symmetric difference of the character sets of T_1 and T_2. Given input trees $T_1, \ldots, T_k$, T is the median tree if it minimizes $\sum_i d(T, T_i)$. The median tree can be computed in polynomial time and has a nice characterization in terms of the character encoding [5, 23, 9]. Both the above notions are related to versions of the local consensus problem (for example, the relaxed versions RV-I and RV-III), and the relevant local consensus trees in many cases contain at least as much "information" as these trees.

The work represented in this paper can be extended in several directions. As we have noted, for all local consensus functions the local consensus tree of a set of k trees can be computed in time polynomial in k and $n = |S|$. Many of these local consensus trees can be constructed in $O(kn)$ time.

REFERENCES

[1] E. Adams III, *N-trees as nestings: Complexity, similarity, and consensus*, J. Classification, 3 (1986), pp. 299–317.

[2] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, MA, 1974.

[3] A. V. Aho, Y. Sagiv, T. G. Szymanski, and J. D. Ullman, *Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions*, SIAM J. Comput., 10 (1981), pp. 405–421.

[4] J. Barthélemy and F. Janowitz, *A formal theory of consensus*, SIAM J. Discrete Math., 3 (1991), pp. 305–322.

[5] J. Barthélemy and F. McMorris, *The median procedure for n-Trees*, J. Classification, 3 (1986), pp. 329–334.

[6] W. Brown, E. M. Prager, A. Wang, and A. C. Wilson, *Mitochondrial DNA sequences of primates: Tempo and mode of evolution*, J. Mol. Evol., 18 (1982), pp. 225–239.

[7] D. Bryant and M. Steel, *Extension operations on sets of leaf-labelled trees*, Research report 118, Department of Mathematics and Statistics, University of Canterbury, Christchurch, New Zealand, 1994.

[8] H. Colonius and H. H. Schulze, *Tree structures for proximity data*, British J. Math. Statist. Psych., 34 (1981), pp. 167–180.

[9] W. H. E. Day, *Optimal algorithms for comparing trees with labeled leaves*, J. Classification, 2 (1985), pp. 7–28.

[10] G. F. Estabrook, C. S. Johnson, Jr., and F. R. McMorris, *An idealized concept of the true cladistic character*, Math. Biosci., 23 (1975), pp. 263–272.

[11] G. F. Estabrook, C. S. Johnson, Jr., and F. R. McMorris, *An algebraic analysis of cladistic characters*, Discrete Math., 16 (1976), pp. 141–147.

[12] G. F. Estabrook, C. S. Johnson, Jr., and F. R. McMorris, *A mathematical foundation for the analysis of cladistic character compatibility*, Math. Biosci., 29 (1976), pp. 181–187.

[13] M. Farach and M. Thorup, *Optimal evolutionary tree comparison by sparse dynamic programming*, in Proc. 35th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Piscataway, NJ, November 1994, pp. 770–779.

[14] M. Farach, T. Przytycka, and M. Thorup, *On the agreement of many trees*, Inform. Process. Lett., 55 (1995), pp. 297–301.

[15] J. Felsenstein, *Numerical methods for inferring evolutionary trees*, Quart. Review of Biology, 57 (1982), pp. 379–404.

[16] C. R. Finden and A. D. Gordon, *Obtaining common pruned trees*, J. Classification, 2 (1985), pp. 225–276.

[17] D. Gusfield, *Efficient algorithms for inferring evolutionary trees*, Networks, 21 (1991), pp. 19–28.

[18] S. Kannan, E. Lawler, and T. Warnow, *Determining the evolutionary tree using experiments*, J. Algorithms, 21 (1996), pp. 26–50.

[19] D. Keselman and A. Amir, *Maximum agreement subtree in a set of evolutionary trees—Metrics and efficient algorithms*, in Proc. 35th Annual Symposium on Foundations of Com-

puter Science, IEEE Computer Society Press, Piscataway, NJ, November 1996, pp. 758–769.
[20] D. Harel and R. Tarjan, *Fast algorithm for finding nearest common ancestors*, SIAM J. Comput., 13 (1984), pp. 338–355.
[21] M. Henzinger, V. King, and T. Warnow, *Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology*, in Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM/SIAM, January 28–30, 1996, pp. 333–340.
[22] G. Nelson, *Cladistic analysis and synthesis: Principles and definitions, with a historical note on Adanson's Famille des Plantes* (1763–1764), Systematic Zoology, 28 (1979), pp. 1–21.
[23] F. McMorris and M. Steel, *The complexity of the median procedure for binary trees*, in Proc. 4th Conference of the International Federation of Classification Societies, Paris, 1993; Stud. Classification Data Anal. Knowledge Organ., by Springer-Verlag, to appear.
[24] M. Steel and T. Warnow, *Kaikoura tree theorems: Computing the maximum agreement subtree*, Inform. Process. Lett., 48 (1993), pp. 77–82.
[25] T. Warnow, *Tree compatibility and inferring evolutionary history*, J. Algorithms, 16 (1994), pp. 388–407.