Fast Compatibility Testing for Rooted Phylogenetic Trees*

Yun Deng[1] **and David Fernández-Baca**[2]

1 **Department of Computer Science, Iowa State University, Ames, IA 50011, USA**
`yundeng@iastate.edu`

2 **Department of Computer Science, Iowa State University, Ames, IA 50011, USA**
`fernande@iastate.edu`

Abstract

We consider the following basic problem in phylogenetic tree construction. Let $\mathcal{P} = \{T_1, \ldots, T_k\}$ be a collection of rooted phylogenetic trees over various subsets of a set of species. The tree compatibility problem asks whether there is a tree T with the following property: for each $i \in \{1, \ldots, k\}$, T_i can be obtained from the restriction of T to the species set of T_i by contracting zero or more edges. If such a tree T exists, we say that $\mathcal{P}$ is compatible.

We give a $\tilde{O}(M_{\mathcal{P}})$ algorithm for the tree compatibility problem, where $M_{\mathcal{P}}$ is the total number of nodes and edges in $\mathcal{P}$. Unlike previous algorithms for this problem, the running time of our method does not depend on the degrees of the nodes in the input trees. Thus, it is equally fast on highly resolved and highly unresolved trees

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory, J.3 Life and Medical Sciences

Keywords and phrases Algorithms, computational biology, phylogenetics

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.12

1 Introduction

Building a phylogenetic tree that encompasses all living species is one of the central challenges of computational biology. Two obstacles to achieving this goal are lack of data and conflict among the data that is available. The data shortage is tied to the vast disparity in the amount of information at our disposal for different families of species and the limited amount of comparable data across families [16]. One approach to overcoming this obstacle begins by identifying subsets of species for which enough data is available, and building phylogenies for each subset. The resulting trees are then synthesized into a single phylogeny – a supertree – for the combined set of species. This approach, proposed in the early 90s [2, 15], has been used successfully to build large-scale phylogenies (see, e.g., [3, 10]).

Any attempt at synthesizing phylogenetic information from multiple input trees must deal with the potential for conflict among these trees. Conflict may arise due to errors, or due to phenomena such as gene duplication and loss, and horizontal gene transfer. A fundamental question is whether conflict exists at all; that is, does there exist a supertree that exhibits the evolutionary relationships implicit in each input tree? We can formalize

* Supported in part by the National Science Foundation under grants CCF-1017189 and CCF-1422134.

this question as follows. Let $\mathcal{P} = \{T_1, \ldots, T_k\}$ be a collection of rooted phylogenetic trees, where, for each $i \in \{1, \ldots, k\}$, T_i is a phylogenetic tree for a set of species $L(T_i)$. The *tree compatibility problem* asks whether there exists a phylogenetic supertree T for the set of species $\bigcup_{i=1}^{k} L(T_i)$ such that, for each $i \in \{1, \ldots, k\}$, T_i can be obtained from $T|L(T_i)$ – the minimal subtree of T spanning $L(T_i)$ – by zero or more contractions of internal edges. If the answer is "yes", then $\mathcal{P}$ is said to be *compatible*; otherwise, $\mathcal{P}$ is *incompatible*.

Here we present an algorithm that solves the compatibility problem for rooted trees in $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time, where $M_{\mathcal{P}}$ is the total number of vertices and edges in the trees in $\mathcal{P}$. This running time is independent of the degrees of the internal nodes of the input trees.

1.1 Previous Work

Aho et al. [1] gave the first polynomial-time algorithm for the rooted tree compatibility problem. Their motivation was not phylogenetics, but relational databases. Steel [18] was perhaps the first to note the relevance of Aho et al.'s algorithm to supertree construction. His version of the Aho et al. algorithm, which he called the `Build` algorithm, has been a major influence in later work, including the present paper.

Henzinger et al. [9] showed that one can check the compatibility of a collection $\mathcal{R}$ of rooted triples – that is, phylogenetic trees on three species – in $O(|\mathcal{R}| \log^2 |\mathcal{R}|)$ time. (The time bound stated in [9] is higher, but can be improved using a faster dynamic graph connectivity data structure [11].) Any collection of trees $\mathcal{P}$ can be encoded by a collection of rooted triples $\mathcal{R}(\mathcal{P})$, obtained by enumerating the restriction of each input tree to every three-element subset of its species set (see Section 2). If n denotes the total number of distinct species in $\mathcal{P}$, then we get a trivial upper bound of $|\mathcal{R}(P)| = O(n^3 k)$. We can improve on this by finding a *minimal* set $\mathcal{R}^*$ of rooted triples that define the input trees. If the trees are binary – *fully resolved*, in the language of phylogenetics –, then $O(n)$ triples suffice for each tree, giving us $|\mathcal{R}^*| = O(nk)$. If input trees admit non-binary – that is, *unresolved* – nodes, however, the number of triples needed per input tree is roughly proportional to n^2 (the precise bound depends on the sum of the products of the degrees of internal nodes and the degrees of their children [8]), giving us $|\mathcal{R}^*| = O(n^2 k)$. Of course, the extra step of finding $\mathcal{R}^*$ adds to the complexity of the algorithm.

The tree compatibility problem is related to the *incomplete directed perfect phylogeny problem* (IDPP). Indeed, any collection of k phylogenetic trees on n distinct species can be encoded as a problem of testing the compatibility of a collection of $O(M_{\mathcal{P}})$ "directed partial characters" on n species[1]. Intuitively, each such character encodes the species in the subtree rooted at some node in an input tree. There is a $\tilde{O}(nm)$ algorithm to test the compatibility of m incomplete characters [14], which can be adapted to yield a $\tilde{O}(nM_{\mathcal{P}})$ algorithm for tree compatibility.

When the input trees are unrooted, the tree compatibility problem becomes NP-hard [18]. Nevertheless, the decision version is polynomial-time solvable if k is fixed [4]; that is, the problem is fixed-parameter tractable in k. The proof of fixed-parameter tractability in [4] relies on Courcelle's Theorem [6], and thus is an existence proof, rather than a practical algorithm.

Finally, we note that there are linear-time algorithms for testing the compatibility of a collection of trees that all have exactly the same leaf label set. One such algorithm can be obtained using recent results on computing "loose" and "strict" consensus trees [13]. Both

[1] For a precise definition of partial characters and IDPP, we refer the reader to Pe'er et al. [14].

types of consensus trees can be found in $O(nk)$ time, which is $O(M_{\mathcal{P}})$ when all leaf label sets are identical. (We thank J. Jansson for pointing this out.)

1.2 Our Contributions

At a high level, our algorithm resembles `Build` [18, 17]. There are, however, important differences. `Build` relies on the *triplet graph*, whose nodes are the species and where there is an edge between two species if they are involved in a triplet (see Section 2). Our algorithm relies instead on intersection graphs of sets of species associated with certain nodes of the input trees. Our graphs allow a more compact representation of the triplets induced by the trees in $\mathcal{P}$ (see Section 3). The key to the correctness of our approach is the close relationship between the triplet graph and our intersection graph (see Lemma 5 of Section 3). We remark that intersection graphs have a long history of use in testing compatibility, beginning with the work of Buneman [5].

We also take ideas from other sources. From Pe'er et al.'s IDPP algorithm [14], we adapt the idea of a *semi-universal node*. Although the graphs used to solve IDPP and rooted compatibility are different, semi-universal nodes play similar roles in each case: they capture the notion of sets of nodes in the input trees that map to the same node in a supertree, if a supertree exists. The relationship between our algorithm and Pe'er et al.'s goes deeper. Our approach can be viewed as an algorithm for IDPP that takes advantage of the fact that our particular set of incomplete characters arises from a collection of trees.

Intersection graphs are a convenient tool to prove the correctness for our algorithm. They are less convenient for an implementation, because they are hard to maintain dynamically, as our algorithm requires. The difficulty lies in recomputing set intersections whenever the graphs are updated. We avoid this by using *display graphs*, an idea that we borrow from the proof of the fixed-parameter tractability of unrooted compatibility [4]. The display graph of a collection $\mathcal{P}$ is obtained by identifying leaves in the input trees that have the same label. Display graphs provide all the connectivity information we need for our intersection graphs (see Lemma 8 of Section 4), but are easier to maintain.

Through our techniques, we achieve what, to our knowledge, is the first algorithm for rooted compatibility to achieve near-linear time under all input conditions, regardless of the degrees of the nodes in the input trees. This is an essential quality for dealing with large datasets.

1.3 Contents

Section 2 reviews basic concepts in phylogenetics, defines compatibility formally, and introduces triplets and the triplet graph. Section 3 presents our intersection graph approach to testing tree compatibility. Section 4 describes the implementation details needed to achieve the $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time bound. Section 5 contains some final remarks.

2 Preliminaries

For each positive integer r, $[r]$ denotes the set $\{1, \ldots, r\}$.

2.1 Phylogenetic Trees

Let T be a rooted tree. We use $V(T)$, $E(T)$, and $r(T)$ to denote the nodes, edges, and the root of T, respectively. For each $x \in V(T)$, we use $\mathrm{Ch}(x)$ and $T(x)$ to denote the set of

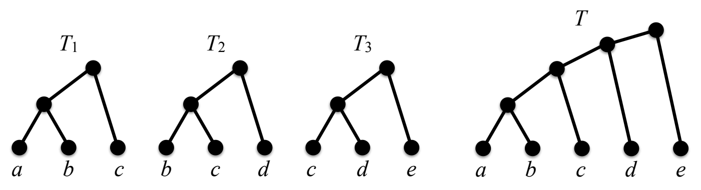

Figure 1 A profile $\mathcal{P} = \{T_1, T_2, T_3\}$ and a tree T that displays $\mathcal{P}$.

children of x and the subtree of T rooted at x, respectively. Suppose $u, v \in V(T)$. Then, u is a *descendant* of v if v lies on the path from u to $r(T)$ in T. Note that v is a descendant of itself. T is *binary*, or *fully resolved*, if each of its internal nodes has two children.

A (rooted) *phylogenetic tree* is a rooted tree T where every internal node has at least two children, along with a bijection λ that maps each leaf of T to an element of a set of *species*, denoted by $L(T)$. For each $x \in V(T)$, $L(x)$ denotes the set of species mapped to the leaves of $T(x)$; that is, $L(x) = \{\lambda(v) : v \text{ is a leaf in } T(x)\}$. $L(x)$ is called the *cluster* at x. Note that $L(r(T)) = L(T)$. The set of all clusters in T is $\text{Cl}(T) = \{L(x) : x \in V(T)\}$.

The following lemma, adapted from [17, p. 52], is part of the folklore of phylogenetics.

▶ **Lemma 1.** *Let $\mathcal{H}$ be a collection of non-empty subsets of a set of species X that includes all singleton subsets of X as well as X itself. If there exists a phylogenetic tree T such that $\text{Cl}(T) = \mathcal{H}$, then, up to isomorphism, T is unique.*

Let T be a phylogenetic tree and A be a set of species. The *restriction* of T to A, denoted $T|A$ is the phylogenetic tree with species set A where $\text{Cl}(T|A) = \{C \cap A : C \in \text{Cl}(T) \text{ and } C \cap A \neq \emptyset\}$. Let T' be a phylogenetic tree. T *displays* T' if $\text{Cl}(T') \subseteq \text{Cl}(T|L(T'))$.

A *rooted triple* is a binary phylogenetic tree on three leaves. A rooted triple with leaves a, b, and c is denoted $ab|c$ if the path from a to b does not intersect the path from c to the root. We treat $ab|c$ and $ba|c$ as equivalent.

When restricted to the three-element subsets of its species set, a phylogenetic tree T induces a set $\mathcal{R}(T)$ of rooted triples, defined as $\mathcal{R}(T) = \{T|X : X \subseteq L(T), |X| = 3 \text{ and } T|X \text{ is binary}\}$.

▶ **Lemma 2** ([17, p. 119])**.** *Let T and T' be two phylogenetic trees. Then T displays T' if and only if $\mathcal{R}(T') \subseteq \mathcal{R}(T)$.*

2.2 Profiles and Compatibility

Throughout the rest of this paper $\mathcal{P} = \{T_1, \dots, T_k\}$ denotes a set where, for each $i \in [k]$, T_i is a phylogenetic tree. We refer to $\mathcal{P}$ as a *profile*, and write $L(\mathcal{P})$ to denote $\bigcup_{i \in [k]} L(T_i)$, the *species set* of $\mathcal{P}$. We write $V(\mathcal{P})$ for $\bigcup_{i \in [k]} V(T_i)$, $E(\mathcal{P})$ for $\bigcup_{i \in [k]} E(T_i)$, and $\mathcal{R}(\mathcal{P})$ for $\bigcup_{i \in [k]} \mathcal{R}(T_i)$. Given a subset A of $L(P)$, $\mathcal{P}|A$ denotes the profile $\{T_1|A, \dots, T_k|A\}$. The *size* of $\mathcal{P}$ is $M_{\mathcal{P}} = |V(\mathcal{P})| + |E(\mathcal{P})|$. Note that $M_{\mathcal{P}} = O(nk)$.

Profile $\mathcal{P}$ is *compatible* if there exists a phylogenetic tree T such that, for each $i \in [k]$, T displays T_i. If such a tree T exists, we say that T *displays* $\mathcal{P}$. See Figure 1.

2.3 The Triplet Graph

The *triplet graph* of a profile $\mathcal{P}$, denoted $\Gamma(\mathcal{P})$, is the graph whose vertex set is $L(P)$ and where there is an edge between species a and b if and only if there exists a $c \in L(P)$ such that $ab|c \in \mathcal{R}(\mathcal{P})$. The following observation concerning singleton profiles will be useful.

▶ **Observation 1.** Let T be a phylogenetic tree with $|L(T)| > 2$. Let $u_1, \dots, u_p$ be the children of $r(T)$. Then, the connected components of $\Gamma(\{T\})$ are $L(u_1), \dots, L(u_p)$, where $p \geq 2$.

3 Testing Compatibility

Here we describe our compatibility algorithm and prove its correctness. We begin with some definitions.

Let U be a subset of $V(\mathcal{P})$ and let $L(U)$ denote $\bigcup_{u \in U} L(u)$. Then, $G_{\mathcal{P}}(U)$ denotes the graph with vertex set U and where $u, v \in U$ are joined by an edge if and only if $L(u) \cap L(v) \neq \emptyset$. That is, $G_{\mathcal{P}}(U)$ is the intersection graph of the clusters associated with the nodes in U. For each $i \in [k]$, let $U(i) = U \cap V(T_i)$. We say that U is *valid* if, for each $i \in [k]$,

V1 if $|U(i)| \geq 2$, then there exists a node $v \in V(T_i)$ such that $U(i) \subseteq \mathrm{Ch}(v)$ and
V2 $L(U(i)) = L(T_i) \cap L(U)$.

Observe that the set U_{init} defined as follows is valid.

$$U_{\text{init}} = \{r(T_i) : i \in [k]\} \tag{1}$$

Note that $L(U_{\text{init}}) = L(\mathcal{P})$. From this point forward, we assume that $G_{\mathcal{P}}(U_{\text{init}})$ is connected. No generality is lost by doing so. To see why, observe that if $G_{\mathcal{P}}(U_{\text{init}})$ is not connected, then $\mathcal{P}$ can be partitioned into a collection of species-disjoint profiles $\mathcal{P}_1, \dots, \mathcal{P}_r$ such that $\mathcal{P}$ is compatible if and only if $\mathcal{P}_j$ is compatible for all $j \in [r]$.

The next observation follows from the definition of a valid set.

▶ **Observation 2.** If U is a valid subset of $V(\mathcal{P})$, then, for each $i \in [k]$, $\mathrm{Cl}(T_i|L(U)) = \{L(U(i))\} \cup \{L(v) : v \text{ is a descendant of a node in } U(i)\}$.

Together with Lemma 1, Observation 2 shows that $T_i|L(U)$ is completely determined by the descendants of $U(i)$.

A valid subset U of $V(\mathcal{P})$ is *compatible* if there exists a phylogenetic tree T with $L(T) = L(U)$ that displays $T_i|L(U)$ for every $i \in [k]$. If such a tree T exists, we say that T *displays* U.

▶ **Lemma 3.** *Profile $\mathcal{P}$ is compatible if and only if every valid subset of $V(\mathcal{P})$ is compatible.*

Proof.
(⇐) If every valid subset of $V(\mathcal{P})$ is compatible, then, in particular, so is the set U_{init} of Equation (1). Let T be a tree that displays U_{init}. Then, $L(T) = L(U_{\text{init}}) = L(\mathcal{P})$. Thus, for every $i \in [k]$, $T_i|L(T) = T_i$, and thus T displays T_i. Hence, $\mathcal{P}$ is compatible.

(⇒) Suppose $\mathcal{P}$ is compatible, but there is a valid subset U of $V(\mathcal{P})$ that is not compatible. Let T be a tree that displays $\mathcal{P}$. But then $T|U$ displays U, a contradiction. ◀

`BuildST` (Algorithm 1), which is closely related to Semple and Steel's `Build` algorithm [17], determines whether a valid set $U \subseteq V(\mathcal{P})$ is compatible. The key difference between `BuildST` and `Build` is that the latter uses the triplet graph $\Gamma(\mathcal{P})$, while `BuildST` uses the graph $G_{\mathcal{P}}(U)$, for different subsets U of $V(\mathcal{P})$. As we show in Lemma 5, the two graphs are closely related. Nevertheless, $G_{\mathcal{P}}(U)$ offers some computational advantages over the triplet graph. Intuitively, this is because $G_{\mathcal{P}}(U)$ is a more compact representation of the triplets in $\mathcal{R}(\mathcal{P})$.

`BuildST`(U) attempts to build a tree T_U for U. Step 1 initializes the root of T_U. If $L(U)$ consists of one or two species, then U is trivially compatible; Steps 2–5 handle these cases.

Algorithm 1: `BuildST`(U)

```
Input: A valid set $U \subseteq V(\mathcal{P})$ such that $G_{\mathcal{P}}(U)$ is connected.
Output: A tree $T_U$ that displays $U$, if $U$ is compatible; incompatible otherwise.
1  Create a node $r_U$
2  if $|L(U)| = 1$ then
3      return the tree consisting of node $r_U$, labeled by the single species in $L(U)$
4  if $|L(U)| = 2$ then
5      return the tree consisting of node $r_U$ and two children, each labeled by a different
        species in $L(U)$
6  foreach $i \in [k]$ such that $|U(i)| = 1$ do
7      Let $v$ be the single element in $U(i)$
8      $U = (U \setminus \{v\}) \cup \mathrm{Ch}(v)$
9  Let $W_1, W_2, \ldots, W_p$ be the connected components of $G_{\mathcal{P}}(U)$
10 if $p = 1$ then
11     return incompatible
12 foreach $j \in [p]$ do
13     Let $t_j$ = BuildST($W_j$)
14     if $t_j$ is a tree then
15         Add $t_j$ to the set of subtrees of $r_U$
16     else
17         return incompatible
18 return the tree with root $r_U$
```

The loop in lines 6–8 identifies the indices $i \in [k]$ such that $U(i)$ is a singleton. For each such i, it removes the single element v in $U(i)$ and replaces v by its children in T_i. Note that if v is a leaf in T_i, then $U(i) = \emptyset$ after this step. As we argue in the proof of Theorem 7, when $\mathcal{P}$ is compatible, all such nodes v – provided they are not leaves – map to the same node w in the tree T that displays $\mathcal{P}$, in the sense that $L(w)$ is the smallest cluster in T such that $L(v) \subseteq L(w)$[2]. In Theorem 7, we also show that, if $G_{\mathcal{P}}(U)$ remains connected after steps 6–8, then U is incompatible. This case is handled in Line 11. Otherwise, Lines 12–17 recursively process each connected component of $G_{\mathcal{P}}(U)$. If the recursive calls succeed in finding trees for all components, these trees are assembled into a phylogeny for U by joining them to the root created in Step 1. If any recursive call determines that a component is incompatible, then U is declared to be incompatible.

The correctness of `BuildST` relies on two lemmas, the first of which can be proved using induction.

▶ **Lemma 4.** *If, given a valid set $U \subseteq V(\mathcal{P})$,* `BuildST`($U$) *returns a tree T_U, then T_U is a phylogenetic tree such that $L(T_U) = L(U)$.*

The next lemma is central to the correctness proof of `BuildST`.

▶ **Lemma 5.** *Let $W_1, \ldots, W_p$ be the connected components of $G_{\mathcal{P}}(U)$ at step 9 of* `BuildST`(U)*, for some valid set $U \subseteq V(\mathcal{P})$. Then,*

(i) *for each $j \in [p]$, W_j is a valid set, and*

(ii) *the connected components of $\Gamma(\mathcal{P}|L(U))$ are precisely $L(W_1), \ldots, L(W_p)$.*

[2] Thus, v plays the role of a *semi-universal node*, in the sense of Pe'er et al. [14].

Proof.

(i) Let U_{bef} and U_{aft} denote the values of U before and after executing steps 6–8. Each element of U_{aft} is either an element of U_{bef} or a child of some $v \in U_{\text{bef}}$. In the latter case, every child of v is in U_{aft}. By assumption, U_{bef} is valid, and for every non-leaf node v, $L(v) = \bigcup_{w \in \text{Ch}(v)} L(w)$; therefore, U_{aft} must also be valid. Part (i) follows.

(ii) We can show that the following holds after steps 6–8.

▶ **Claim 6.** *Let a and b be any two species in $L(U)$. Then, (a, b) is an edge in $\Gamma(\mathcal{P}|L(U))$ if and only if there exists a node $v \in U$ such that $a, b \in L(v)$.*

Observe that both $\Pi_1 = \{A : A \text{ is a connected component of } \Gamma(\mathcal{P}|L(U))\}$ and $\Pi_2 = \{L(W) : W \text{ is a connected component of } G_{\mathcal{P}}(U)\}$ are partitions of $L(U)$. We prove that $\Pi_1 = \Pi_2$ by showing that (a) for each connected component A of $\Gamma(\mathcal{P}|L(U))$ there exists a connected component W of $G_{\mathcal{P}}(U)$ such that $A \subseteq L(W)$, and (b) for each connected component W of $G_{\mathcal{P}}(U)$ there exists a connected component A of $\Gamma(\mathcal{P}|L(U))$ such that $L(W) \subseteq A$.

(a) Let A be any connected component of $\Gamma(\mathcal{P}|L(U))$. We argue that any two species a, b in A must be in the same connected component of $G_{\mathcal{P}}(U)$. Let $U_a = \{v \in U : a \in L(v)\}$ and $U_b = \{v \in U : b \in L(v)\}$. Then, each of U_a and U_b is a clique in $G_{\mathcal{P}}(U)$. It thus suffices to show that there is a path between some node in U_a and some node in U_b. By the definition of A, there exists a path between a and b in $\Gamma(\mathcal{P}|L(U))$. Suppose this path is $\rho = \langle a_1, \dots, a_m \rangle$, where $a_1 = a$ and $a_m = b$. By Claim 6, for each $l \in [m-1]$, there exists a node $w_l \in U$ such that $\{a_l, a_{l+1}\} \subseteq L(w_l)$. For each $l \in [m-2]$, $L(w_l) \cap L(w_{l+1}) \neq \emptyset$, so, either $w_l = w_{l+1}$ or there is a edge between w_l and w_{l+1} in $G_{\mathcal{P}}(U)$. Let $\pi = \langle w_1, \dots, w_{m-1} \rangle$. Then, we can extract from π a subsequence that is a path from w_1 to w_{m-1} in $G_{\mathcal{P}}(U)$. By the definition of ρ, $a \in L(w_1)$ and $b \in L(w_{m-1})$, so $w_1 \in U_a$ and $w_l \in U_b$. This completes the proof of part (a).

(b) Let W be any connected component of $G_{\mathcal{P}}(U)$. If $|L(W)| = 1$, the statement holds trivially, so assume that $|L(W)| > 1$. We argue that any two species a, b in $L(W)$ are in the same connected component of $\Gamma(\mathcal{P}|L(U))$. Let v_a and v_b be nodes in W such that $a \in L(v_a)$ and $b \in L(v_b)$. If $v_a = v_b$, then, by Claim 6, (a, b) is an edge of $\Gamma(\mathcal{P}|L(U))$, and we are done. So, suppose instead that $v_a \neq v_b$.

Let us call a path π from v_a to v_b *good* if $|L(w)| > 1$ for every node w in π. We claim that there exists a good path from v_a to v_b. To prove this claim, we first argue that we can choose v_a and v_b such that $|L(v_a)|, |L(v_b)| > 1$. Indeed, consider the case of species a (the case for b is analogous). If $|L(v)| = 1$ for every node $v \in W$ such that $a \in L(v)$, then we would have $|L(W)| = 1$, contradicting our assumption that $|L(W)| > 1$. Now, suppose the path π from v_a to v_b has a node $w \notin \{v_a, v_b\}$ such that $|L(w)| = 1$. Let w' and w'' be the predecessor and successor of w in π. Then, $L(w') \cap L(w'') = L(w) \neq \emptyset$, so there is an edge between w' and w''. Thus, we can delete w from π and the resulting sequence remains a path between v_a and v_b.

Let $\pi = \langle w_1, \dots, w_l \rangle$, where $w_1 = v_a$ and $w_l = v_b$, be a good path from v_a to v_b in $G_{\mathcal{P}}(U)$. Choose a sequence of species $\rho = \langle c_1, \dots, c_{l+1} \rangle$, where $c_1 = a$, $c_{l+1} = b$ and, for each $j \in [l]$, $c_j, c_{j+1} \in L(w_j)$ and $c_j \neq c_{j+1}$. Note that such a choice is always possible. Then, by Claim 6, (c_j, c_{j+1}) is an edge of $\Gamma(\mathcal{P}|L(U))$. Hence, ρ is a path from a to b in $\Gamma(\mathcal{P}|L(U))$. ◀

We are now ready to prove the correctness of `BuildST`.

▶ **Theorem 7.** *Let U_{init} be the set defined in Equation (1). Then,* `BuildST`(U_{init}) *either (i) returns a tree T that displays $\mathcal{P}$, if $\mathcal{P}$ is compatible, or (ii) returns* ***`incompatible`*** *otherwise.*

Proof. We first argue that if `BuildST`(U_{init}) outputs `incompatible`, $\mathcal{P}$ is indeed incompatible. Assume, on the contrary, that $\mathcal{P}$ is compatible. Then, there must be a call `BuildST`(U) for some valid subset U such that $|L(U)| > 2$, in which the graph $G(U)$ of step 9 has a single connected component, $W_1 = U$. By Lemma 3, U must be compatible, so there exists a phylogeny T_U that displays U. By Observation 1, $\Gamma(\{T_U\})$ has at least two connected components A and B. By Lemma 5(ii), however, $\Gamma(\mathcal{P}|L(U))$ is connected, so there exist species $a \in A$ and $b \in B$ such that $ab|c \in \mathcal{R}(\mathcal{P}|U)$. But $ab|c \notin \mathcal{R}(T)$, and, by Lemma 2, T does not display some tree in $\mathcal{P}|L(U)$, a contradiction. Thus, $G(U)$ has at least two components.

Now, suppose that `BuildST`(U_{init}) returns a tree T. We prove that T displays $\mathcal{P}$ by arguing that for each $i \in [k]$ there is an injective mapping $\phi_i : V(T_i) \to V(T)$ that maps every node $v \in V(T_i)$ to a distinct node $\phi_i(v) \in V(T)$ such that $L(v) \subseteq L(\phi_i(v))$.

By Lemma 4, each recursive call `BuildST`(U) returns a phylogenetic tree T_U for $L(U)$. Let r_U denote the root of T_U. We have two cases.

Case (i): $|L(U)| \leq 2$. For each $i \in [k]$, we must have $|U(i)| \in \{0, 1, 2\}$; we only need to consider $|U(i)| \in \{1, 2\}$. Suppose first that $|U(i)| = 1$, and let v be the single node in $U(i)$. Note that $L(v) \subseteq L(r_U)$. Thus, we make $\phi_i(v) = r_U$. If $|L(U(i))| = 1$, we are done. Otherwise, $|L(U(i))| = 2$. Then, v has two children, v_1 and v_2, both leaves, labeled with, say, species s_1 and s_2, respectively. Node r_U also has two children, r_1 and r_2. Assume, without loss of generality, that these children are labeled with species s_1 and s_2, respectively. Then, $L(v_j) = L(r_j)$ for $j \in \{1, 2\}$. Therefore, we make $\phi_i(v_j) = r_j$ for each $j \in \{1, 2\}$. Now, suppose that $|U(i)| = 2$. Then, $|L(U(i))| = 2$, and each node in $U(i)$ is a leaf in T_i. As in the previous case, we map each node of $U(i)$ to the corresponding child of r_U.

Case (ii): $|L(U)| > 2$. Let U_{bef} be the value of U before entering the loop of lines 6–8, and let U_{aft} be the value of U at line 9, after the loop of lines 6–8 terminates. Let $U_{\text{rem}} = \{v \in U_{\text{bef}} : v \in U_{\text{bef}}(i) \text{ for some } i \in [k] \text{ such that } |U_{\text{bef}}(i)| = 1\}$. Then $U_{\text{aft}} = (U_{\text{bef}} \setminus U_{\text{rem}}) \cup \{u \in \text{Ch}(v) : v \in U_{\text{rem}}\}$. Assume inductively that every descendant of a node in U_{aft} is mapped to an appropriate node in T_U. It therefore suffices to establish mappings for the nodes in U_{rem}. Now, for every $v \in U_{\text{rem}}$, $L(v) \subseteq L(r_U)$. Thus, we make $\phi(v) = r_U$ for every $v \in U_{\text{rem}}$. ◀

4 Implementation

We now explain how to implement `BuildST` in order to solve the tree compatibility problem in $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time. Consider a call to `BuildST`(U). Recall that we can assume that $G_{\mathcal{P}}(U)$ is connected. `BuildST`(U) requires the following three pieces of information.

(G1) *The value of* $|L(U)|$. This number is needed in Lines 2 and 4 of `BuildST`.

(G2) *The set* $J(U)$ *of all* $i \in [k]$ *such that* $|U(i)| = 1$. Set $J(U)$ contains the indices i considered in Lines 6–8 of `BuildST`.

(G3) *The set* $U(i) = U \cap V(T_i)$ *for each* $i \in [k]$. For each $i \in J(U)$, $U(i)$ contains precisely the element v used in Lines 7 and 8 of `BuildST`.

It is straightforward to obtain (G1), (G2), and (G3) for the valid set U_{init} of Equation (1): $|L(U_{\text{init}})| = n$, $J(U_{\text{init}}) = [k]$, and, for every $i \in [k]$, $U_{\text{init}}(i) = \{r(T_i)\}$. Now assume that we have (G1), (G2), and (G3) at the beginning of some call to `BuildST`(U). Steps 6–8 modify U and, therefore, $G_{\mathcal{P}}(U)$. Suppose that, at Line 9, $G_{\mathcal{P}}(U)$ has more than one connected component. We need to compute (G1), (G2), and (G3) for each connected component, in order to pass this information to the recursive calls in Line 13. That is, if $p > 1$, for each $j \in [p]$, we need to compute $|L(W_j)|$, $J(W_j)$, and $W_j(i) = W_j \cap V(T_i)$, for each $i \in [k]$.

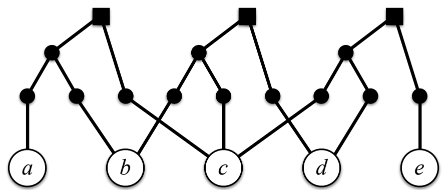

Figure 2 The graph $H_{\mathcal{P}}(U_{\text{init}})$ for the profile $\mathcal{P}$ of Figure 1. Nodes of U_{init} are drawn as squares. Nodes in the set $\{x_s : s \in L(\mathcal{P})\}$ are labeled with the corresponding species. Species labeling the leaves of trees in $\mathcal{P}$ are omitted.

We use the dynamic graph connectivity data structure by Holm et al. [11]. We refer to this data structure as *HDT*. HDT allows us to maintain the list of nodes in each component, as well as the number of these nodes so that, if we start with no edges in a graph with N vertices, the amortized cost of each update is $O(\log^2 N)$. For efficiency, however, we do not use HDT directly on $G_{\mathcal{P}}(U)$. The reason is that the edges of $G_{\mathcal{P}}(U)$ are defined via intersections of sets of species, which could make it costly to determine the new nodes and edges created as a result of Step 8. To avoid this problem, we proceed indirectly, through an auxiliary graph $H_{\mathcal{P}}(U)$, defined below. As we shall see, $H_{\mathcal{P}}(U)$ offers another advantage over $G_{\mathcal{P}}(U)$: maintaining $H_{\mathcal{P}}(U)$ only requires handling deletions, but maintaining $G_{\mathcal{P}}(U)$ additionally requires handling insertions.

We define $H_{\mathcal{P}}(U)$ as a subgraph of the graph $H_{\mathcal{P}}$ constructed as follows. For each species $s \in L(\mathcal{P})$, create a new node $x_s \notin V(\mathcal{P})$, and let $X_{\mathcal{P}} = \{x_s : s \in L(\mathcal{P})\}$. Then, $H_{\mathcal{P}}$ is the graph whose vertex set is $V(\mathcal{P}) \cup X_{\mathcal{P}}$ and whose edge set is $E(\mathcal{P}) \cup \{(u, x_s) : u$ is a leaf in T_i, for some $i \in [k]$, such that $\lambda(u) = s\}$. Note that $H_{\mathcal{P}}$ has $O(M_{\mathcal{P}})$ nodes and edges, and can be constructed from $\mathcal{P}$ in $O(M_{\mathcal{P}})$ time. $H_{\mathcal{P}}$ is essentially the *display graph* for $\mathcal{P}$ [4]. The display graph is the result of glueing together leaves in $\mathcal{P}$ labeled by the same species. Contrast this with $H_{\mathcal{P}}$, which connects leaves with a common label through nodes in $X_{\mathcal{P}}$. This minor difference with respect to the display graph serves to simplify our presentation.

Given a valid subset U of $V(\mathcal{P})$, we define $H_{\mathcal{P}}(U)$ as the subgraph of $H_{\mathcal{P}}$ induced by $\{v : v$ is a descendant of some node $u \in U\} \cup \{x_s \in X_{\mathcal{P}} : s \in L(U)\}$. Note that $H_{\mathcal{P}}(U_{\text{init}}) = H_{\mathcal{P}}$. See Figure 2.

The next result states the basic properties of $H_{\mathcal{P}}(U)$. Due to space limitations, we omit its proof.

▶ **Lemma 8.** *The following statements hold for any valid subset U of $V(\mathcal{P})$.*

(i) *Let v be a node in U. If $U' = (U \setminus \{v\}) \cup \mathrm{Ch}(v)$, then $H_{\mathcal{P}}(U')$ is obtained from $H_{\mathcal{P}}(U)$ by deleting v and every edge (v, u) such that $u \in \mathrm{Ch}(v)$.*

(ii) *Any two nodes in U are in the same connected component in $G_{\mathcal{P}}(U)$ if and only if they are in the same connected component of $H_{\mathcal{P}}(U)$.*

By Lemma 8(ii), the connected components $W_1, \dots, W_p$ of $G_{\mathcal{P}}(U)$ can be put into a one-to-one correspondence with the connected components $Y_1, \dots, Y_p$ of $H_{\mathcal{P}}(U)$ so that $W_j = Y_j \cap U$ for each $j \in [p]$.

We represent $H_{\mathcal{P}}(U)$ using the aforementioned HDT data structure. For each connected component Y of $H_{\mathcal{P}}(U)$, we maintain three fields:

(H1) Y.`count`, the cardinality of $Y \cap X_{\mathcal{P}}$,

(H2) Y.`singleton`, a doubly-linked list that contains all indices $i \in [k]$ such that $|U(i)| = 1$, and

(H3) Y.`List`, an array where, for each $i \in [k]$, Y.`List`$[i]$ is a doubly-linked list consisting of the elements of $Y \cap U(i)$.

Recall that we assume that $G_{\mathcal{P}}(U)$ is connected at the beginning of a call to `BuildST`(U). Thus, by Lemma 8, $H_{\mathcal{P}}(U)$ has a single connected component, Y. Then, $|L(U)| = Y$.`count`, $J(U) = Y$.`singleton`, and Y.`List`$[i]$ contains the elements of $U(i)$, for each $i \in [k]$. Thus, the three fields of Y provide `BuildST`(U) with the information that it needs – that is, (G1), (G2), and (G3). In particular, they allow us to easily find each node v considered in Line 7 of `BuildST`(U). Line 8 is then performed as a series of edge deletions, one for each edge (v, u) such that $u \in \mathrm{Ch}(v)$, followed by the deletion of v (we provide further details below). By Lemma 8(i), this correctly updates $H_{\mathcal{P}}(U)$. The deletions break up $H_{\mathcal{P}}(U)$ into a collection of connected components $Y_1, \dots, Y_p$. For each $j \in [p]$, Y_j corresponds to a connected component W_j of $G_{\mathcal{P}}(U)$ that (if $p > 1$) is processed in a recursive call in Line 13. We need to compute Y_j.`count`, Y_j.`singleton`, Y_j.`List` for each $j \in [p]$, in order to provide this information to the recursive calls.

The total number of edge and node deletions executed by `BuildST`(U_{init}) – including all deletions conducted by the recursive calls – cannot exceed the total number of edges and nodes in $H_{\mathcal{P}}$, which is $O(M_{\mathcal{P}})$. The HDT data structure allows us to maintain connectivity information throughout the entire algorithm in $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$. In the remainder of this section, we show that we can maintain the `count`, `singleton`, and `List` fields throughout the entire algorithm in total time $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$. We also argue that all the required information for $H_{\mathcal{P}}(U_{\text{init}})$ can be initialized in $O(M_{\mathcal{P}})$ time.

Let $Y_{\text{init}} = V(\mathcal{P}) \cup X_{\mathcal{P}}$ be the vertex set of $H_{\mathcal{P}}(U_{\text{init}})$. Then, Y_{init} is the single connected component of $H_{\mathcal{P}}(U_{\text{init}})$. We initialize the data fields of Y_{init} as follows: (1) Y_{init}.`count` $= |L(\mathcal{P})|$, (2) Y_{init}.`singleton` is the set $[k]$, and (3) for each $i \in [k]$, Y_{init}.`List`$[i]$ consists of $r(T_i)$. Thus, we can initialize all data fields in $O(M_{\mathcal{P}})$ time.

We assume that every node v in $H_{\mathcal{P}}(U)$ is either *marked*, if $v \in U$, or *unmarked*, if $v \notin U$. Initially, each node $v \in U_{\text{init}}$ is marked, and every node $v \in Y_{\text{init}} \setminus U_{\text{init}}$ is unmarked. We also assume that for each node v in $H_{\mathcal{P}}(U)$, we maintain sufficient information to determine in $O(1)$ time whether $v \in X_{\mathcal{P}}$ or $v \in V(\mathcal{P})$, and that, in the latter case, we have $O(1)$-time access to the index $i \in [k]$ such that $v \in V(T_i)$. For each i such that Y.`List`$[i]$ contains exactly one element, we maintain a pointer from Y.`List`$[i]$ to the entry for i in Y.`singleton`. This allows us to update Y.`singleton` in $O(1)$ time when $U(i)$ is no longer a singleton. For each marked node $v \in Y$ (so $v \in U$), we maintain a pointer from v to the element in Y.`List`$[i]$ that contains v. This allows us to update Y.`List`$[i]$ in $O(1)$ time when v becomes unmarked.

Consider a call to `BuildST`(U) for some valid set U. Step 1 takes $O(1)$ time. Since $H_{\mathcal{P}}(U)$ initially consists of a single connected component, say Y, and we have Y.`count`, Steps 2–5 also take $O(1)$ time. Let $H = H_{\mathcal{P}}(U)$. We implement the loop in lines 6–8 as follows. First, we enumerate the indices in $J = J(U)$ in $O(|J|)$ time by listing the elements of Y.`singleton`. For each $i \in J$, we retrieve and remove the single element v_i of $U(i)$ from Y.`List`$[i]$, and then delete i from Y.`singleton`. This takes $O(1)$ time. We unmark v_i, and for every node $u \in \mathrm{Ch}(v_i)$ we mark u and add it to Y.`List`$[i]$. This takes $O(1)$ time per edge. We then successively delete each edge (v_i, u) such that $u \in \mathrm{Ch}(v_i)$, updating (H1)–(H3) for each newly-created component along the way. Once these edges are deleted, we delete v_i itself. By Lemma 8(i), the result is the graph $H_{\mathcal{P}}(U)$ for the new set U. Let us focus on how to handle the deletion of a single edge $e = (v_i, u)$.

Let Y' be the connected component of H that currently contains v_i. We query the HDT data structure to determine, in $O(\log^2 M_{\mathcal{P}})$ amortized time, whether deleting (v_i, u) splits Y' into two components. If Y' remains connected, no updates are needed. Otherwise, Y' is split into two parts Y_1 and Y_2. To fill in the `count`, `singleton`, and `List` fields of Y_1 and Y_2, we use the well-known technique of scanning the smaller component [7]. We query the HDT data structure to determine, in $O(1)$ time, which of Y_1 and Y_2 has fewer nodes. Suppose without loss of generality that $|Y_1| \le |Y_2|$. We initialize Y_2.`count` and Y_2.`List` to Y'.`count` and Y'.`List`, respectively. We initialize Y_1.`count` to 0 and Y_1.`List`$[i]$ to null for each $i \in [k]$. We then scan each node v in Y_1, and do the following. If $v \in X_{\mathcal{P}}$, we decrement Y_2.`count` and increment Y_1.`count`. Otherwise $v \in V(\mathcal{P})$; assume that $v \in V(T_i)$. If v is marked, we remove v from Y_2.`List`$[i]$ and add v to Y_1.`List`$[i]$; each such move takes $O(1)$ time. This operation requires at most one update in each of Y_1.`singleton` and Y_2.`singleton`; each update takes $O(1)$ time.

We claim that any node v is scanned $O(\log M_{\mathcal{P}})$ times over the entire execution of `BuildST`(U_{init}). To verify this, let $N(v)$ be the number of nodes in the connected component containing v. Suppose that, initially, $N(v) = N$. Then, the rth time we scan v, $N(v) \le N/2^r$. Thus, v is scanned $O(\log N)$ times. The claim follows, since $N = O(M_{\mathcal{P}})$. Therefore, the total number of updates over all nodes is $O(M_{\mathcal{P}} \log M_{\mathcal{P}})$, and the work per update is $O(1)$.

To summarize, the work done by `BuildST` consists of three parts: (i) initialization, (ii) maintaining connected components, and (iii) maintaining the `count`, `singleton`, and `List` fields for each connected component. Part (i) takes $O(M_{\mathcal{P}})$ time. Part (ii) involves $O(M_{\mathcal{P}})$ edge and node deletions on the HDT data structure, at an amortized cost of $O(\log^2 M_{\mathcal{P}})$ per deletion. Part (iii) involves scanning the nodes of our graph every time a deletion creates a new component, for a total of $O(M_{\mathcal{P}} \log M_{\mathcal{P}})$ scans, at $O(1)$ cost per scan, over the entire execution of `BuildST`. This yields our main result.

▶ **Theorem 9.** *Let U_{init} be the set defined in Equation* (1)*. Then, there exists and implementation of* `BuildST` *such that* `BuildST`(U_{init}) *runs in $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time.*

5 Discussion

A trivial lower bound for the tree compatibility problem is $\Omega(M_{\mathcal{P}})$, the time to read the input. Thus, our result leaves us a polylogarithmic factor away from an optimal algorithm for compatibility. Is it possible to reduce or even eliminate this gap? The bottleneck is the time to maintain the information associated with the various components of $H_{\mathcal{P}}(U)$. It is conceivable that the special structure of this graph and the way the deletions are performed could be used to our advantage. A second question is how well our algorithm performs in practice. To investigate this, it should be possible to leverage existing knowledge on the empirical behavior of dynamic connectivity data structures [12].

References

1 Alfred V. Aho, Yehoshua Sagiv, Thomas G. Szymanski, and Jeffrey D. Ullman. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM J. Comput.*, 10(3):405–421, 1981.

2 Bernard R. Baum. Combining trees as a way of combining data sets for phylogenetic inference, and the desirability of combining gene trees. *Taxon*, 41:3–10, 1992.

3 Olaf R. P. Bininda-Emonds, Marcel Cardillo, Kate E. Jones, Ross D. E. MacPhee, Robin M. D. Beck, Richard Grenyer, Samantha A. Price, Rutger A. Vos, John L. Gittleman, and Andy Purvis. The delayed rise of present-day mammals. *Nature*, 446:507–512, 2007.

4 David Bryant and Jens Lagergren. Compatibility of unrooted phylogenetic trees is FPT. *Theoretical Computer Science*, 351:296–302, 2006.

5 Peter Buneman. A characterisation of rigid circuit graphs. *Discrete Math.*, 9:205–212, 1974.

6 Bruno Courcelle. The monadic second-order logic of graphs I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.

7 Shimon Even and Yossi Shiloach. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, January 1981. URL: `http://doi.acm.org/10.1145/322234.322235`, `doi:10.1145/322234.322235`.

8 Stefan Grünewald, Mike Steel, and M. Shel Swenson. Closure operations in phylogenetics. *Mathematical Biosciences*, 208:521–537, 2007.

9 Monika Rauch Henzinger, Valerie King, and Tandy Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24:1–13, 1999.

10 Cody E. Hinchliff, Stephen A. Smith, James F. Allman, J. Gordon Burleigh, Ruchi Chaudhary, Lyndon M. Coghill, Keith A. Crandall, Jiabin Deng, Bryan T. Drew, Romina Gazis, Karl Gude, David S. Hibbett, Laura A. Katz, H. Dail Laughinghouse IV, Emily Jane McTavish, Peter E. Midford, Christopher L. Owen, Richard H. Reed, Jonathan A. Reesk, Douglas E. Soltis, Tiffani Williams, and Karen A. Cranston. Synthesis of phylogeny and taxonomy into a comprehensive tree of life. *Proceedings of the National Academy of Sciences*, 2015. In press. `doi:10.1073/pnas.1423041112`.

11 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, July 2001. URL: `http://doi.acm.org/10.1145/502090.502095`, `doi:10.1145/502090.502095`.

12 Raj Iyer, David Karger, Hariharan Rahul, and Mikkel Thorup. An experimental study of polylogarithmic, fully dynamic, connectivity algorithms. *J. Exp. Algorithmics*, 6, December 2001. URL: `http://doi.acm.org/10.1145/945394.945398`, `doi:10.1145/945394.945398`.

13 Jesper Jansson, Chuanqi Shen, and Wing-Kin Sung. Improved algorithms for constructing consensus trees. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1800–1813, 2013. URL: `http://dx.doi.org/10.1137/1.9781611973105.129`, `doi:10.1137/1.9781611973105.129`.

14 Itsik Pe'er, Tal Pupko, Ron Shamir, and Roded Sharan. Incomplete directed perfect phylogeny. *SIAM J. Comput.*, 33(3):590–607, 2004. URL: `http://dx.doi.org/10.1137/S0097539702406510`, `doi:10.1137/S0097539702406510`.

15 Mark A. Ragan. Phylogenetic inference based on matrix representation of trees. *Molecular Phylogenetics and Evolution*, 1:53–58, 1992.

16 Michael J. Sanderson. Phylogenetic signal in the eukaryotic tree of life. *Science*, 321(5885):121–123, 2008.

17 Charles Semple and Mike Steel. *Phylogenetics.* Oxford Lecture Series in Mathematics. Oxford University Press, Oxford, 2003.

18 Mike A. Steel. The complexity of reconstructing trees from qualitative characters and subtrees. *J. Classification*, 9:91–116, 1992.