RESEARCH

An efficient algorithm for testing the compatibility of phylogenies with nested taxa

Yun Deng† and David Fernández-Baca*†

Abstract

Background: Semi-labeled trees generalize ordinary phylogenetic trees, allowing internal nodes to be labeled by higher-order taxa. Taxonomies are examples of semi-labeled trees. Suppose we are given collection \( P \) of semi-labeled trees over various subsets of a set of taxa. The ancestral compatibility problem asks whether there is a semi-labeled tree that respects the clusterings and the ancestor/descendant relationships implied by the trees in \( P \). The running time and space usage of the best previous algorithm for testing ancestral compatibility depend on the degrees of the nodes in the trees in \( P \).

Results: We give an algorithm for the ancestral compatibility problem that runs in \( O(M_P \log^2 M_P) \) time and uses \( O(M_P) \) space, where \( M_P \) is the total number of nodes and edges in the trees in \( P \).

Conclusions: Taxonomies enable researchers to expand greatly the taxonomic coverage of their phylogenetic analyses. The running time of our method does not depend on the degrees of the nodes in the trees in \( P \). This characteristic is important when taxonomies—which can have nodes of high degree—are used.

Keywords: Algorithms, Phylogenetics, Supertrees, Taxonomies

Introduction

In the tree compatibility problem, we are given a collection \( P = \{T_1, T_2, \ldots, T_k\} \) of rooted phylogenetic trees with partially overlapping taxon sets. \( P \) is called a profile and the trees in \( P \) are the input trees. The question is whether there exists a tree \( T \) whose taxon set is the union of the taxon sets of the input trees, such that \( T \) exhibits the clusterings implied by the input trees. That is, if two taxa are together in a subtree of some input tree, then they must also be together in some subtree of \( T \). The tree compatibility problem has been studied for over three decades [1–4].

In the original version of the tree compatibility problem, only the leaves of the input trees are labeled. Here we study a generalization, called ancestral compatibility, in which taxa may be nested. That is, the internal nodes may also be labeled; these labels represent higher-order taxa, which are, in effect, sets of taxa. Thus, for example, an input tree may contain the taxon \( Glycine \ max \) (soybean) nested within a subtree whose root is labeled Fabaceae (the legumes), itself nested within an Angiosperm subtree. Note that leaves themselves may be labeled by higher-order taxa. The question now is whether there is a tree \( T \) whose taxon set is the union of the taxon sets of the input trees, such that \( T \) exhibits not only the clusterings among the taxa, but also the ancestor/descendant relationships among taxa in the input trees. Our main result is an \( O(M_P \log^2 M_P) \) algorithm for the compatibility problem for trees with nested taxa, where \( M_P \) is the total number of nodes and edges in the trees in \( P \).

Background

The tree compatibility problem is a basic special case of the supertree problem. A supertree method is a way to synthesize a collection of phylogenetic trees with partially overlapping taxon sets into a single supertree that represents the information in the input trees. The supertree approach, proposed in the early 90s [5, 6], has been used successfully to build large-scale phylogenies [7].

The original supertree methods were limited to input trees where only the leaves are labeled. Page [8] was...
among the first to note the need to handle phylogenies where internal nodes are labeled, and taxa are nested. A major motivation is the desire to incorporate taxonomies as input trees in large-scale supertree analyses, as way to circumvent one of the obstacles to building comprehensive phylogenies: the limited taxonomic overlap among different phylogenetic studies [9]. Taxonomies group organisms according to a system of taxonomic rank (e.g., family, genus, and species); two examples are the NCBI taxonomy [10] and the Angiosperm taxonomy [11]. Taxonomies spanning a broad range of taxa provide a collection of semi-labeled trees is more complex than that of a collection of ordinary phylogenies, we are able to extend several of the key ideas—notably, that of a semi-universal label—to the general setting of semi-labeled trees. As in [2], the implementation relies on a dynamic graph data structure, but it requires a more careful amortized analysis based on a weighing scheme.

Contents
This paper has five sections, in addition to this introduction. The section titled "Preliminaries" presents basic definitions regarding graphs, semi-labeled trees, and ancestral compatibility. The section titled "The display graph" introduces the display graph and discusses its properties. The section titled "Testing ancestral compatibility" presents BuildNT, our algorithm for testing ancestral compatibility. We first present the algorithm recursively, and then show how to transform it into an iterative algorithm, BuildNTI, that is easier to implement. We also give an example of the execution of BuildNTI. The "Implementation" section gives the implementation details for BuildNTI. The "Discussion" section gives some concluding remarks.

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

Graph notation
Let \( G \) be a graph. \( V(G) \) and \( E(G) \) denote the node and edge sets of \( G \). The degree of a node \( v \in V(G) \) is the number of edges incident on \( v \). A tree is an acyclic connected graph. In this paper, all trees are assumed to be rooted. For a tree \( T \), \( r(T) \) denotes the root of \( T \). Suppose \( u, v \in V(T) \). Then, \( u \) is an ancestor of \( v \) in \( T \), denoted \( u \leq_T v \), if \( u \) lies on the path from \( v \) to \( r(T) \) in \( T \). If \( u \leq_T v \), then \( v \) is a descendant of \( u \). Node \( u \) is a proper descendant of \( v \) if \( u \) is a descendant of \( v \) and \( u \neq v \). If \( \{u, v\} \in E(T) \) and \( u \leq_T v \), then \( u \) is the parent of \( v \) and \( v \) is a child of \( u \). If neither \( u \leq_T v \) nor \( v \leq_T u \) hold, then we write \( u \parallel_T v \) and say that \( u \) and \( v \) are not comparable in \( T \).

Semi-labeled trees
A semi-labeled tree is a pair \( T = (T, \phi) \) where \( T \) is a tree and \( \phi \) is a mapping from a set \( L(T) \) to \( V(T) \) such that, for every node \( v \in V(T) \) of degree at most two,
\( v \in \phi(L(T)) \), \( L(T) \) is the label set of \( T \) and \( \phi \) is the labelling function of \( T \).

For every node \( v \in V(T) \), \( \phi^{-1}(v) \) denotes the (possibly empty) subset of \( L(T) \) whose elements map into \( v \); these elements as the labels of \( v \) (thus, each label is a taxon). If \( \phi^{-1}(v) \neq \emptyset \), then \( v \) is labeled; otherwise, \( v \) is unlabeled.

Note that, by definition, every leaf in a semi-labeled tree is labeled. Further, any node, the root, that has a single child must be labeled. Nodes with two or more children may be labeled or unlabeled. A semi-labeled tree \( T = (T, \phi) \) is singularly labeled if every node in \( T \) has at most one label; \( T \) is fully labeled if every node in \( T \) is labeled.

Semi-labeled trees, also known as \( X \)-trees, generalize ordinary phylogenetic trees, also known as phylogenetic \( X \)-trees [14]. An ordinary phylogenetic tree is a semi-labeled tree \( T = (T, \phi) \) where \( \phi(T) \) has degree at least two and \( \phi \) is a bijection from \( L(T) \) into leaf set of \( T \) (thus, internal nodes are not labeled).

Let \( T = (T, \phi) \) be a semi-labeled tree and let \( \ell \) and \( \ell' \) be two labels in \( L(T) \). If \( \phi(\ell) \leq_T \phi(\ell') \), then we write \( \ell \leq_T \ell' \) and say that \( \ell' \) is a descendant of \( \ell \) in \( T \) and that \( \ell \) is an ancestor of \( \ell' \). We write \( \ell \prec_T \ell' \) if \( \phi(\ell) \) is a proper descendant of \( \phi(\ell') \). If \( \phi(\ell) \parallel_T \phi(\ell') \), then we write \( \ell \parallel_T \ell' \) and say that \( \ell \) and \( \ell' \) are not comparable in \( T \). If \( T \) is fully labeled and \( \phi(\ell) \) is the parent of \( \phi(\ell') \) in \( T \), then \( \ell \) is the parent of \( \ell' \) in \( T \) and \( \ell' \) is a child of \( \ell \) in \( T \); two labels with the same parent are siblings.

Two semi-labeled trees \( T = (T, \phi) \) and \( T' = (T', \phi') \) are isomorphic if there exists a bijection \( \psi : V(T) \rightarrow V(T') \) such that \( \phi' = \psi \circ \phi \) and, for any two nodes \( u, v \in V(T) \), \( \langle u, v \rangle \in E(T) \) if and only if \( \langle \psi(u), \psi(v) \rangle \in E(T') \).

Let \( T = (T, \phi) \) be a semi-labeled tree. For each \( u \in V(T) \), \( X(u) \) denotes the set of all labels in the subtree of \( T \) rooted at \( u \); that is, \( X(u) = \bigcup_{v \in S(T, u)} \phi^{-1}(v) \). \( X(u) \) is called a cluster of \( T \). \( Cl(T) \) denotes the set of all clusters of \( T \). It is well known [14, Theorem 3.5.2] that a semi-labeled tree \( T \) is completely determined by \( Cl(T) \). That is, if \( Cl(T) = Cl(T') \) for some other semi-labeled tree \( T' \), then \( T \) is isomorphic to \( T' \).

Suppose \( A \subseteq L(T) \) for a semi-labeled tree \( T = (T, \phi) \). The restriction of \( T \) to \( A \), denoted \( T|A \), is the semi-labeled tree whose cluster set is \( Cl(T|A) = \{ X \cap A : X \in Cl(T) \text{ and } X \cap A \neq \emptyset \} \). Intuitively, \( T|A \) is obtained from the minimal rooted subtree of \( T \) that connects the nodes in \( \phi(A) \) by suppressing all vertices of degree two that are not in \( \phi(A) \).

Let \( T = (T, \phi) \) and \( T' = (T', \phi') \) be semi-labeled trees such that \( L(T') \subseteq L(T) \). \( T \) ancestrally displays \( T' \) if \( Cl(T') \subseteq Cl(T|L(T')) \). Equivalently, \( T \) ancestrally displays \( T' \) if \( T' \) can be obtained from \( T|L(T') \) by contracting edges, and, for any \( \ell_1, \ell_2 \in L(T') \),

- (i) if \( \ell_1 \prec_T \ell_2 \), then \( \ell_1 \prec_T \ell_2 \), and
- (ii) if \( \ell_1 \parallel_T \ell_2 \), then \( \ell_1 \parallel_T \ell_2 \).

The notion of “ancestrally displays” for semi-labeled trees generalizes the well-known notion of “displays” for ordinary phylogenetic trees [14].

For a semi-labelled tree \( T \), let us define \( D(T) \) and \( N(T) \) as follows.

\[
D(T) = \{(\ell, \ell') : \ell, \ell' \in L(T) \text{ and } \ell \prec_T \ell'\}
\]
\[
N(T) = \{(\ell, \ell') : \ell, \ell' \in L(T) \text{ and } \ell \parallel_T \ell'\}
\]

Note that \( D(T) \) consists of ordered pairs, while \( N(T) \) consists of unordered pairs.

**Lemma 1** (Bordewich et al. [13]) Let \( T \) and \( T' \) be semi-labelled trees such that \( L(T') \subseteq L(T) \). Then \( T \) ancestrally displays \( T' \) if and only if \( D(T') \subseteq D(T) \) and \( N(T') \subseteq N(T) \).

**Profiles and ancestral compatibility**

Throughout the rest of this paper \( \mathcal{P} = \{T_1, T_2, \ldots, T_k\} \) denotes a set where, for each \( i \in [k] \), \( T_i = (T_i, \phi_i) \) is a semi-labeled tree. We refer to \( \mathcal{P} \) as a profile, and write \( \mathcal{L}(\mathcal{P}) \) to denote \( \bigcup_{i \in [k]} L(T_i) \), the label set of \( \mathcal{P} \). Figure 1 shows a profile where \( \mathcal{L}(\mathcal{P}) = \{a, b, c, d, e, f, g, h, i\} \). We write \( V(\mathcal{P}) \) for \( \bigcup_{i \in [k]} V(T_i) \) and \( E(\mathcal{P}) \) for \( \bigcup_{i \in [k]} E(T_i) \). The size of \( \mathcal{P} \) is \( |V(\mathcal{P})| + |E(\mathcal{P})| \).

\( \mathcal{P} \) is ancestrally compatible if there is a rooted semi-labeled tree \( T \) that ancestrally displays each of the trees in \( \mathcal{P} \). If \( T \) exists, we say that \( T \) ancestrally displays \( \mathcal{P} \) (see Fig. 2).

Given a subset \( X \) of \( L(T) \), the restriction of \( \mathcal{P} \) to \( X \), denoted \( \mathcal{P}|X \), is the profile defined as

\[
\mathcal{P}|X = \{T_1|X \cap L(T_1), T_2|X \cap L(T_2), \ldots, T_k|X \cap L(T_k)\}
\]

The proof of the following lemma is straightforward.

**Lemma 2** Suppose \( \mathcal{P} \) is ancestrally compatible and let \( T \) be a tree that ancestrally displays \( \mathcal{P} \). Then, for any \( X \subseteq L(\mathcal{P}), T|X \text{ ancestrally displays } \mathcal{P}|X \).

---

![Fig. 1](image-url) A profile \( \mathcal{P} = \{T_1, T_2, T_3\} \)—trees are ordered left-to-right. The letters are the original labels; grey numbers are labels added to make the trees fully labeled (Adapted from [16])
For technical reasons, fully labeled trees are easier to handle than those that are not. Suppose \( \mathcal{P} \) contains trees that are not fully labeled. We can convert \( \mathcal{P} \) into an equivalent profile \( \mathcal{P}' \) of fully-labeled trees as follows. For each \( i \in [k] \), let \( l_i \) be the number of unlabeled nodes in \( T_i \). Create a set \( L' \) of \( n' = \sum_{i \in [k]} l_i \) labels such that \( L' \cap L(\mathcal{P}) = \emptyset \). For each \( i \in [k] \) and each \( v \in V(T_i) \) such that \( \phi_i^{-1}(v) = \emptyset \), make \( \phi_i^{-1}(v) = \{ l \} \), where \( l \) is a distinct element from \( L' \). We refer to \( \mathcal{P}' \) as the profile obtained by adding distinct new labels to \( \mathcal{P} \) (see Fig. 1).

**Lemma 3** (Daniel and Semple [15]) Let \( \mathcal{P}' \) be the profile obtained by adding distinct new labels to \( \mathcal{P} \). Then, \( \mathcal{P} \) is ancestrally compatible if and only if \( \mathcal{P}' \) is ancestrally compatible. Further, if \( \mathcal{T} \) is a semi-labeled phylogenetic tree that ancestrally displays \( \mathcal{P}' \), then \( \mathcal{T} \) ancestrally displays \( \mathcal{P} \).

From this point forward, we make the following assumption.

**Assumption 1** For each \( i \in [k] \), \( T_i \) is fully and singularly labeled.

By Lemma 3, no generality is lost in assuming that all trees in \( \mathcal{P} \) are fully labeled. The assumption that the trees are singularly labeled is inessential; it is only for clarity. Note that, even with the latter assumption, a tree that ancestrally displays \( \mathcal{P} \) is not necessarily singularly labeled. Figure 2 illustrates this fact.

The display graph

The display graph of a profile \( \mathcal{P} \), denoted \( H_\mathcal{P} \), is the graph obtained from the disjoint union of the underlying trees \( T_1, T_2, \ldots, T_k \) by identifying nodes that have the same label. Multiple edges between the same pair of nodes are replaced by a single edge. See Fig. 3.

\( H_\mathcal{P} \) has \( O(M_\mathcal{P}) \) nodes and edges, and can be constructed in \( O(M_\mathcal{P}) \) time. By Assumption 1, there is a bijection between the labels in \( L(\mathcal{P}) \) and the nodes of \( H_\mathcal{P} \). Thus, from this point forward, we refer to the nodes of \( H_\mathcal{P} \) by their labels. It is easy to see that if \( H_\mathcal{P} \) is not connected, then \( \mathcal{P} \) decomposes into label-disjoint subprofiles, and that \( \mathcal{P} \) is compatible if and only if each subprofile is compatible. Thus, without loss of generality, we shall assume the following.

**Assumption 2** \( H_\mathcal{P} \) is connected.

Positions

Our compatibility algorithm processes the trees in \( \mathcal{P} \) from the top down, starting at the roots. We refer to the set of nodes in \( \mathcal{P} \) currently being considered as a “position”. The algorithm advances from the current position to the next by replacing certain nodes in the current position by their children. Formally, a position (for \( \mathcal{P} \)) is a vector \( U = (U(1), U(2), \ldots, U(k)) \), where \( U(i) \subseteq L(T_i) \), for each \( i \in [k] \). Since labels may be shared among trees, we may have \( U(i) \cap U(j) \neq \emptyset \), for \( i, j \in [k] \) with \( i \neq j \). For each \( i \in [k] \), let \( \text{Desc}_i(U) = \{ \ell : \ell' \leq_{T_i} \ell, \text{ for some } \ell' \in U(i) \} \), and let \( \text{Desc}_\mathcal{P}(U) = \bigcup_{i \in [k]} \text{Desc}_i(U) \).

A position \( U \) is valid if, for each \( i \in [k] \),

- (V1) if \( |U(i)| \geq 2 \), then the elements of \( U(i) \) are siblings in \( T_i \)
- (V2) \( \text{Desc}_i(U) = \text{Desc}_\mathcal{P}(U) \cap L(T_i) \).

**Lemma 4** For any valid position \( U \),

\[
\mathcal{P}\rhd \text{Desc}_\mathcal{P}(U) = [T_1|\text{Desc}_1(U), T_2|\text{Desc}_2(U), \ldots, T_k|\text{Desc}_k(U)].
\]

**Proof** By (V2), we have that \( T_i|\text{Desc}_i(U) \) and \( T_j|\text{Desc}_\mathcal{P}(U) \cap L(T_i) \) are isomorphic, for each \( i \in [k] \). The lemma then follows from the definition of \( \mathcal{P}\rhd \text{Desc}_\mathcal{P}(U) \).

For any valid position \( U \), \( H_\mathcal{P}(U) \) denotes the subgraph of \( H_\mathcal{P} \) induced by \( \text{Desc}_\mathcal{P}(U) \).
Observation 1 For any valid position \( U \), \( H_\mathcal{P}(U) \) is the subgraph of \( H_\mathcal{P} \) obtained by deleting all labels in \( V(H_\mathcal{P}) \) \( \setminus \text{Desc}_\mathcal{P}(U) \), along with all incident edges.

A valid position of special interest to us is \( U_{\text{root}} \), the root position, defined as follows.

\[
U_{\text{root}} = (\phi_i^{-1}(r(T_1)), \phi_j^{-1}(r(T_2)), \ldots, \phi_k^{-1}(r(T_k))).
\]

That is, for each \( i \in [k] \), \( U_{\text{root}}(i) \) is a singleton containing only the label of \( r(T_i) \). In Fig. 3, \( U_{\text{root}} = ((1), (2), (g)) \). It is straightforward to verify that \( U_{\text{root}} \) is indeed valid, that \( \text{Desc}_\mathcal{P}(U_{\text{root}}) = L(\mathcal{P}) \), and that \( H_\mathcal{P}(U_{\text{root}}) = H_\mathcal{P} \).

**Semi-universal labels**

Let \( U \) be a valid position, and let \( \ell \) be a label in \( U \). Then, \( \ell \) is semi-universal in \( U \) if \( U(i) = \{\ell\} \), for every \( i \in [k] \) such that \( \ell \in \mathcal{L}(T_i) \). In Fig. 3, labels 1 and 2 are semi-universal in \( U_{\text{root}} \), but 3 is not, since 3 is in both \( L(T_2) \) and \( L(T_3) \), but \( U_{\text{root}}(2) \neq \{g\} \).

The term “semi-universal”, borrowed from Pe'er et al. [19], derives from the following fact. Suppose that \( \mathcal{P} \) is ancestrally compatible, that \( T \) is a tree that ancestrally displays \( \mathcal{P} \), and that \( \ell \) is a semi-universal label for some valid position \( U \). Then, as we shall see, \( \ell \) must label the root \( u_\ell \) of a subtree of \( T \) that contains all the descendants of \( \ell \) in \( T_i \) for every \( i \) such that \( \ell \in \mathcal{L}(T_i) \). The qualifier “semi” is because this subtree may also contain labels that do not descend from \( \ell \) in any input tree, but descend instead from some other semi-universal label \( \ell' \) in \( U \). In this case, \( \ell' \) also labels \( u_\ell \). We exploit this property of semi-universal labels in our ancestral compatibility algorithm and its proof of correctness (see “Testing ancestral compatibility”).

For each label \( \ell \in L(\mathcal{P}) \), let \( k_\ell \) denote the number of input trees that contain label \( \ell \). We can obtain \( k_\ell \) for every \( \ell \in L(\mathcal{P}) \) in \( O(M_\mathcal{P}) \) time during the construction of \( H_\mathcal{P} \).

**Lemma 5** Let \( U = (U(1), \ldots, U(k)) \) be a valid position. Then, label \( \ell \) is semi-universal in \( U \) if the cardinality of the set \( J_\ell = \{i \in [k] : U(i) = \{\ell\}\} \) equals \( k_\ell \).

**Proof** By definition, \( U(i) = \{\ell\} \), for every \( i \in J_\ell \). Since \( |J_\ell| = k_\ell \), the lemma follows. \( \square \)

**Successor positions**

For every \( i \in [k] \) and every \( \ell \in L(T_i) \), let \( \text{Ch}_i(\ell) \) denote the set of children of \( \ell \) in \( L(T_i) \). For a subset \( A \) of \( L(T_i) \), let \( \text{Ch}_i(A) = \bigcup_{\ell \in A} \text{Ch}_i(\ell) \). Let \( U \) be a valid position, and \( S \) be the set of semi-universal labels in \( U \). The successor of \( U \) with respect to \( S \) is the position \( U'(i) = (U'(1), U'(2), \ldots, U'(k)) \), where, for each \( i \in [k] \), \( U'(i) \) is defined as follows.

\[
U'(i) = \begin{cases} 
\text{Ch}_i(\ell) & \text{if } U(i) = \{\ell\}, \text{ for some } \ell \in S, \\
U(i) & \text{otherwise}.
\end{cases}
\]

In Fig. 3, the set of semi-universal labels in \( U_{\text{root}} \) is \( S = \{1, 2\} \). Since \( \text{Ch}_1(1) = \{3, f\} \) and \( \text{Ch}_2(2) = \{e, f, g\} \), the successor of \( U_{\text{root}} \) is \( U'_{\text{root}} = \{(3, f), \{e,f,g\}, \{g\} \} \).

Observation 2 Let \( U \) be a valid position, and let \( U' \) be the successor of \( U \) with respect to the set \( S \) of semi-universal labels in \( U \). Then, \( H_\mathcal{P}(U') \) can be obtained from \( H_\mathcal{P}(U) \) by doing the following for each \( \ell \in S \): (1) for each \( i \in [k] \) such that \( U(i) = \{\ell\} \), delete all edges between \( \ell \) and \( \text{Ch}_i(\ell) \); (2) delete \( \ell \).

Let \( U \) be a valid position, and \( W \) be a subset of \( \text{Desc}_\mathcal{P}(U) \). Then, \( U|W \) denotes the position \( (U(1) \cap W, U(2) \cap W, \ldots, U(k) \cap W) \). In Fig. 3, the components of \( H_\mathcal{P}(U') \), where \( U' \) is the successor of \( U_{\text{root}} \), are \( W_1 = \{3, 4, a, b, c, d, e, g\} \) and \( W_2 = \{f, h, i\} \). Thus, \( U'|W_1 = [(3), \{e,g\}, \{g\}] \) and \( U'|W_2 = [(f), \{f\}, \emptyset] \). We have the following result.

**Lemma 6** Let \( U \) be a valid position, and \( S \) be the set of all semi-universal labels in \( U \). Let \( U' \) be the successor of \( U \) with respect to \( S \), and let \( W_1, W_2, \ldots, W_p \) be the label sets of the connected components of \( H_\mathcal{P}(U') \). Then, \( U'|W_j \) is a valid position, for each \( j \in [p] \).

**Proof** It suffices to argue that \( U' \) satisfies conditions (VI) and (V2). The lemma then follows from the fact that the connected components of \( H_\mathcal{P}(U') \) are label-disjoint.

\( U' \) must satisfy condition (VI), since \( U \) does. Suppose \( \ell \in S \). Then, for each \( i \in [k] \) such that \( \ell \in \mathcal{L}(T_i) \), \( \text{Desc}_i(U') = \text{Desc}_i(U) \setminus \{\ell\} \) and \( \text{Desc}_i(U') \cap L(T_i) = (\text{Desc}_i(U) \cap L(T_i)) \setminus \{\ell\} \). Thus, since (V2) holds for \( U \), it also holds for \( U' \). \( \square \)

**Testing ancestral compatibility**

**Overview of the algorithm**

BuildNT (Algorithm 1) is our algorithm for testing compatibility of semi-labeled trees. Its argument, \( U \), is a valid position in \( \mathcal{P} \) such that \( H_\mathcal{P}(U) \) is connected. BuildNT relies on the fact—proved later, in Theorem 1—that if \( \mathcal{P}|\text{Desc}_\mathcal{P}(U) \) is compatible, then \( U \) must contain a non-empty set \( S \) of semi-universal labels. If such a set \( S \) exists, the algorithm replaces \( U \) by its successor \( U' \) with respect to \( S \). It then processes each connected component of \( H_\mathcal{P}(U') \) recursively, to determine if the associated sub-profile is compatible. If all the recursive calls are successful, then their results are combined into a super tree for \( \mathcal{P}|\text{Desc}_\mathcal{P}(U) \).
In detail, BuildNT proceeds as follows. Line 1 computes the set $S$ of semi-universal labels in $U$. If $S$ is empty, then, $P|\text{Desc}_P(U)$ is incompatible, and, thus, so is $P$. This fact is reported in Line 3. Line 4 creates a tentative root $r_U$, labeled by $S$, for the tree $T_U$ for $L(U)$. Line 5 checks if $S$ contains exactly one label $\ell$, with no proper descendants. If so, by the connectivity assumption, $\ell$ must be the sole member of $\text{Desc}_P(U)$ that is, $L(U) = \ell$. Therefore, Line 6 simply returns the tree with a single node, labeled by $S = \{\ell\}$. Line 7 updates $U$, replacing it by its successor. Line 8 checks if the set $S$ of semi-universal labels obtained in Line 1 is empty. By Lemma 6, $U|W_j$ is a valid position, for each $j \in [p]$. Lines 8–12 recursively invoke BuildNT on $U|W_j$ for each $j \in [p]$, to determine if there is a tree $t_j$ that ancestrally displays $P|\text{Desc}_P(U \cap W_j)$. If any subproblem is incompatible, Line 12 reports that $P$ is incompatible. Otherwise, Line 13 returns the tree obtained by making the $t_j$s the subtrees of root $r_U$.

Thus, we can assume inductively that $t_j$ is a phylogenetic tree for $L(W_j)$. Since $S \cup \bigcup_{j \in [p]} L(W_j) = L(U)$, the tree returned in Line 13 is a phylogeny with species set $L(U)$.

\begin{algorithm}
\small
\textbf{Input:} A valid position $U$ for $P$ such that $H_P(U)$ is connected.
\textbf{Output:} A semi-labeled tree that ancestrally displays $P'$ = $P|\text{Desc}_P(U)$, if $P'$ is an acceptably compatible; incompatible otherwise.
1. Let $S = \{\ell \in U : \ell$ is semi-universal in $U\}$
2. if $S = \emptyset$ then
3. \hspace{0.5em} return incompatible
4. Create a node $r_U$ with label set $S$
5. if $|S| = 1$ and the single element of $S$ has no proper descendants then
6. \hspace{0.5em} return $r_U$
7. Replace $U$ by the successor of $U$ with respect to $S$
8. Let $W_1, W_2, \ldots, W_p$ be the connected components of $H_P(U)$
9. for each $j \in [p]$ do
10. \hspace{0.5em} Let $t_j = \text{BuildNT}(U|W_j)$
11. if $t_j$ is not a tree then
12. \hspace{0.5em} \hspace{0.5em} return incompatible
13. return the tree with root $r_U$ and subtrees $t_1, \ldots, t_p$
\end{algorithm}

Next, we argue the correctness of BuildNT.

\section*{Correctness}

\begin{lemma}
Let $U$ be a valid position in $P$. If BuildNT($U$) returns a tree $T_U$, then $T_U$ is a phylogenetic tree such that $L(T_U) = L(U)$.
\end{lemma}

\begin{proof}
We use induction on $|L(U)|$. The base case, where $|L(U)| = 1$, is handled by Lines 5–6. In this case, $S = L(U) = \{\ell\}$ and BuildNT($U$) correctly returns the tree consisting of a single node, labeled by $\{\ell\}$. Otherwise, let $W_1, \ldots, W_p$ be the connected components of $H_P(U)$ in step 8. Since BuildNT($U$) returns tree $T_U$, it must be the case that, for each $j \in [p]$, the result $t_j$ returned by the recursive call to BuildNT($U|W_j$) in Line 10 is a tree. Since $|S| \geq 1$, we have $|L(W_j)| < |L(U)|$, for each $j \in [p]$. Each $i \in [k]$. Consider any $(\ell, \ell') \in D(T_i)$. Then, $\ell$ has a child $\ell''$ in $T_i$ such that $\ell'' \leq T_i \ell'$—note that we may have $\ell'' = \ell$. There must be a recursive call to BuildNT($U_i$), for some valid position $U_i$ where $\ell$ is the set $S$ of semi-universal labels obtained in Line 1. By Observation 2, label $\ell''$, and thus $\ell'$, both lie in one of the connected components of the graph obtained by deleting all labels in $S$, including $\ell$, and their incident edges from $H_P(U_i)$. It now follows from the construction of $T_i$ that $(\ell, \ell') \in D(T_i)$. Thus, $D(T_i) \subseteq D(T_i)$. Now, consider any $(\ell, \ell') \in N(T_i)$. Let $v$ be the lowest common ancestor of $\phi_i(\ell)$ and $\phi_i(\ell')$ in $T_i$ and let $t_v$ be the label of $v$. Then, $t_v$ has a group of children, $\ell_1$ and $\ell_2$, say, in $T_i$ such that $\ell_1 \leq T_i \ell$, and $\ell_2 \leq T_i \ell'$. Because BuildNT($U_{\text{root}}$) returns a tree, there are recursive calls BuildNT($U_1$) and BuildNT($U_2$) for valid positions $U_1$ and $U_2$ such that $\ell_1$ is semi-universal for $U_1$ and $\ell_2$ is semi-universal for $U_2$. We must have $U_1 \neq U_2$ other-
wise, \(|U_1(i)| = |U_2(i)| \geq 2\), and, thus, neither \(\ell_1\) nor \(\ell_2\) is semi-universal, a contradiction. Further, it follows from the construction of \(T\) that we must have \(\text{Desc}_T(U_1) \cap \text{Desc}_T(U_2) = \emptyset\). Hence, \(\ell \not\in T \ell'\), and, therefore, \((\ell, \ell') \in N(T)\).

(ii) Assume, by way of contradiction, that \(\text{BuildNT}(U_{\text{root}})\) returns \text{incompatible}, but that \(P\) is ancestrally compatible. By assumption, there exists a semi-labeled tree \(T\) that ancestrally displays \(P\). Since \(\text{BuildNT}(U_{\text{root}})\) returns \text{incompatible}, there is a recursive call to \(\text{BuildNT}(U)\) for some valid position \(U\) such that \(U\) has no semi-universal label, and the set \(S\) of Line 1 is empty. By Lemma 2, \(T|\text{Desc}_T(U)\) ancestrally displays \(P|\text{Desc}_T(U)\). Thus, by Lemma 4, \(T|\text{Desc}_T(U)\) ancestrally displays \(T_i|\text{Desc}_i(U)\), for every \(i \in [k]\). Let \(\ell\) be any label in the label set of the root of \(T|\text{Desc}_T(U)\). Then, for each \(i \in [k]\) such that \(\ell \in L(T_i)\), \(\ell\) must be the label of the root of \(T_i|\text{Desc}_i(U)\). Thus, for each such \(i\), \(U(i) = \{\ell\}\). Hence, \(\ell\) is semi-universal in \(U\), a contradiction.

\[\square\]

**Algorithm 2: BuildNT\(_N\)(\(P\))**

**Input:** A profile \(P\).

**Output:** A tree \(T\) that displays \(P\), if \(P\) is compatible; \text{incompatible} otherwise.

1. \(\text{Construct } H_P(U_{\text{root}})\).
2. \(\text{ENQUEUE}(Q, (U_{\text{root}}, \text{null}))\).
3. \(\text{while } Q \text{ is not empty do}\)
4.   \(\langle U, \text{pred} \rangle = \text{DBQUEUE}(Q)\).
5.   \(\text{Let } S = \{\ell \in U : \ell \text{ is semi-universal in } U\}\).
6.   \(\text{if } S = \emptyset \text{ then}\)
7.     \(\text{return } \text{incompatible}\).
8.   \(\text{Create } r_U \text{ with label set } S \text{ and set } \text{parent}(r_U) = \text{pred}\).
9.   \(\text{if } |S| = 1 \text{ and the single element of } S \text{ has no proper descendants then}\)
10.   \(\text{continue}\).
11. \(\text{Replace } U \text{ by the successor of } U \text{ with respect to } S\).
12. \(W_1, W_2, \ldots , W_p \text{ be the connected components of } H_P(U)\).
13. \(\text{foreach } j \in [p] \text{ do}\)
14.   \(\text{ENQUEUE}(Q, (U|W_j, r_U))\).
15. \(\text{return the tree with root } r_{U_{\text{root}}}\).

**An iterative version**

We now present BuildNT\(_N\) (Algorithm 2), an iterative version of BuildNT, which lends itself naturally to an efficient implementation. BuildNT\(_N\) performs a breadth-first traversal of BuildNT’s recursion tree, using a first-in-first-out queue \(Q\) that stores pairs of the form \((U, \text{pred})\), where \(U\) is a valid position in \(P\) and \(\text{pred}\) is a reference to the parent of the node corresponding to \(U\) in the supertree built so far. BuildNT\(_N\) simulates recursive calls in BuildNT by enqueuing pairs corresponding to subproblems. We explain this in more detail next.

BuildNT\(_N\) initializes its queue to contain the starting position, \(U_{\text{root}}\), with a null parent. It then proceeds to the \text{while} loop of Lines 3–14. Each iteration of the loop starts by dequeuing a valid position \(U\), along with a reference \(\text{pred}\) to the potential parent for the subtree for \(L(U)\) in the supertree. The body of the loop closely follows the steps performed by a call to BuildNT\(_N\)(\(U\)). Line 5 computes the set \(S\) of semi-universal labels in \(U\). If \(S\) is empty, the algorithm reports that \(P\) is incompatible and terminates (Lines 6–7). The algorithm then creates a tentative root \(r_U\) labeled by \(S\) for the tree \(T_U\) for \(L(U)\), and links \(r_U\) to its parent (Line 8). If \(S\) consists of exactly one element that has no proper descendants, we skip the rest of the current iteration of the \text{while} loop, and \text{continue} to the next iteration (Lines 9–10). Line 11 replaces \(U\) by its successor with respect to \(S\). Lines 13–14 enqueue each of \(U|W_1, U|W_2, \ldots , U|W_p\) along with \(r_U\), for processing in a subsequent iteration. If the \text{while} loop terminates without any incompatibility being detected, the algorithm returns the tree with root \(r_{U_{\text{root}}}\).

Although the order in which BuildNT\(_N\) processes connected components differs from that of BuildNT — breadth-first instead of depth-first —, it is straightforward.

to see that the effect is equivalent, and the proof of correctness of BuildNT (Theorem 1) applies to BuildNT\(_N\) as well. We thus state the following without proof.

**Theorem 2**  Let \(P = \{T_1, T_2, \ldots , T_k\}\) be a profile. Then, BuildNT\(_N\)(\(P\)) returns either (i) a semi-labeled tree \(T\) that ancestrally displays \(P\), if \(P\) is ancestrally compatible, or (ii) \text{incompatible} otherwise.

Let \(Q\) be BuildNT\(_N\)’s first-in-first-out queue. In the rest of the paper, we will say that a valid position \(U\) is in \(Q\) if \((U, \text{pred}) \in Q\), for some \(\text{pred}\). Let \(H_Q\) be the subgraph of \(H_P\) induced by \(\bigcup\{\text{Desc}(U) : U \text{ is in } Q\}\). By Observation 1, \(H_Q\) is obtained from \(H_P\) through edge and node deletions.
Lemma 8 At the start of any iteration of BuildNTN’s while loop, the set of connected components of $H_Q$ is $\{V(H_P(U)) : U \text{ is in } Q\}$.

Proof The property holds at the outset, since, by Assumption 2, $H_P = H_P(U_{\text{root}})$ is a connected graph, and the only element of $Q$ is $(U_{\text{root}}, \text{null})$. Assume that the property holds at the beginning of iteration $l$. Let $(U, \text{pred})$ be the element dequeued from $Q$ in Line 4. Then, $H_P(U)$ is connected. In place of $(U, \text{pred})$, Lines 13–14 enqueue $(U|W_j,r_{U})$, for each $j \in [p]$, where, by construction, $H_P(U|W_j)$ is a connected component of $H_P(U)$. Thus, the property holds at the beginning of iteration $l + 1$.

In other words, Lemma 8 states that each iteration of BuildNTN($P$) deals with a subgraph of $H_P$, whose connected components are in one-to-one correspondence with the valid positions stored in $Q$. This is illustrated by the next example.

An example

Figures 4, 5, 6, 7 and 8 illustrate the execution of BuildNTN on the profile $P = (T_1, T_2, T_3)$ of Fig. 1. The figures show how the graph $H_Q$—initially equal to $H_P = H_P(U_{\text{root}})$ (Fig. 3)—evolves as its edges and nodes are deleted.

In each figure, $H_Q$ is shown on the left and the current supertree is shown on the right. For brevity, the figures only exhibit the state of $H_Q$ and the supertree after all the nodes at each level are generated. The various valid positions processed by BuildNTN($P$) are denoted by $U_\alpha$, for different subscripts $\alpha$. $S_\alpha$ denotes the semi-universal labels in $U_\alpha$ and $U'_\alpha$ denotes the ancestor of $U_\alpha$ with respect to $S_\alpha$. We write $L_\alpha$ as an abbreviation for $L(U_\alpha)$. The root of the tree for $L_\alpha$ is $r_{U_\alpha}$ and is labeled by $S_\alpha$.

Initially, $Q = ((U_{\text{root}}, \text{null}))$. In what follows, the elements of $Q$ are listed from front to rear.

Level 0. Refer to Fig. 4. As seen earlier, the set of semi-universal labels of $U_{\text{root}}$ is $S_{\text{root}} = \{1, 2\}$. Thus, $H_P(U_{\text{root}})$ has two components $W_1$ and $W_2$. Let $U_1 = U_{\text{root}}|W_1$ and $U_2 = U_{\text{root}}|W_2$. Then,

$U_1 = ([3], \{e,g\}, \{g\})$ and $U_2 = ([f], \{f\}, \emptyset)$.

After level 0 is processed, $Q = ((U_{11}, r_{U_1}), (U_{12}, r_{U_2}))$. Then, the roots of the subtrees for $L_1$ and $L_2$ will be children of $r_{U_{\text{root}}}$.

Level 1. Refer to Fig. 5. We have $S_{11} = \{3\}$, so $H_P(U_{11}')$ has two components $W_{11}$ and $W_{12}$. Let $U_{11} = U_{11}'|W_{11}$ and $U_{12} = U_{11}'|W_{12}$. Then,

$U_{11} = ([a,d], \{g\}, \{g\})$ and $U_{12} = ([e], \{e\}, \emptyset)$.

We have $S_{12} = \{f\}$, so $H_P(U_{12}')$ has two components $W_{21}$ and $W_{22}$. Let $U_{21} = U_{12}'|W_{21}$ and $U_{22} = U_{12}'|W_{22}$. Then,

$U_{21} = (\emptyset, \{h\}, \emptyset)$ and $U_{21} = (\emptyset, \{i\}, \emptyset)$.

After level 1 is processed, $Q = ((U_{111}, r_{U_{11}}), (U_{112}, r_{U_{12}}), (U_{121}, r_{U_{12}}), (U_{122}, r_{U_{12}}))$.

Level 2. Refer to Fig. 6. We have $S_{111} = \{g\}$, so $H_P(U_{111}')$ has two components $W_{111}$ and $W_{112}$. Let $U_{111} = U_{111}'|W_{111}$ and $U_{112} = U_{111}'|W_{112}$. Then,

$U_{111} = ([a], \{a\}, \{4\})$ and $U_{112} = ([d], \{d\}, \{d\})$.

The only semi-universal labels in $U_{121}, U_{211}$, and $U_{221}$ are, respectively, $e$, $h$, and $i$. Since none of these labels have proper descendants, each of them is a leaf in the supertree.

After level 2 is processed, $Q = ((U_{111}, r_{U_{11}}), (U_{112}, r_{U_{12}}), (U_{121}, r_{U_{12}}), (U_{122}, r_{U_{12}}))$.

Level 3. Refer to Fig. 7. We have $S_{111} = \{4, a\}$, so $H_P(U_{111}')$ has two components $W_{1111}$ and $W_{1112}$. Let $U_{1111} = U_{111}'|W_{1111}$ and $U_{1112} = U_{111}'|W_{1112}$. Then,

$U_{1111} = ([b], \emptyset, \{b\})$ and $U_{1112} = ([c], \emptyset, \{c\})$. 
The only semi-universal label in $U_{112}$ is $d$. Since $d$ has no proper descendants, it becomes a leaf in the supertree. After level 3 is processed, $Q = ((U_{1111}, r_{1111}), (U_{1112}, r_{1112})).$

Level 4. Refer to Fig. 8. The only semi-universal labels in $U_{1111}$ and $U_{1112}$ are, respectively, $b$ and $c$. Since neither of these labels have proper descendants, each of them is a leaf in the supertree.

After level 4 is processed, $Q$ is empty, and $\text{BuildNTN}(\mathcal{P})$ terminates.

**Implementation**

Here we prove the following result.

**Theorem 3** There is an algorithm that, given a profile $\mathcal{P}$ of rooted trees, runs in $O(M \log^2 M \mathcal{P})$ time, and either returns a tree that displays $\mathcal{P}$, if $\mathcal{P}$ is compatible, or reports that is $\mathcal{P}$ is incompatible otherwise.

We prove this theorem by showing how to implement $\text{BuildNTN}$ so that the algorithm runs in $O(M \log^2 M \mathcal{P})$ on any profile $\mathcal{P}$.

As in the section titled "An iterative version", let $H_Q$ denote the subgraph of $H_\mathcal{P}$ associated with the valid positions in $\text{BuildNTN}'$s queue. By Lemma 8, each valid position $U$ in $Q$ corresponds to one connected component of $H_Q$ —namely $\text{Desc}(U)$ —and vice-versa. We use this fact in the implementation of $\text{BuildNTN}$: alongside each valid position $U$ in $Q$, we also store a reference to the respective connected component, together with additional information, described next, to quickly identify semi-universal labels.

Let $U$ be any valid set in $Q$, let $Y = V(H_\mathcal{P}(U))$ be the corresponding connected component of $H_Q$, and let $\ell$ be any label in $Y$. Our implementation maintains the following data fields:

- Let $J_U = \{i \in [k] : U(i) \neq \emptyset\}$. Then, $Y.\text{map}$ is a map from $J_U$ to $L(U)$, where, for each $i \in J_U$, $Y.\text{map}(i) = U(i)$.
- For each $\ell \in Y$, $\ell.\text{count}$ equals the cardinality of the set $\{i \in [k] : Y.\text{map}(i) = \ell\}$. (Recall that $k_\ell$ is the number of input trees that contain $\ell$.)
- $Y.\text{exposed}$, a set consisting of all $i \in [k]$ such that $Y.\text{map}(i) = \ell$ for some $\ell \in Y$ such that $\ell.\text{count} = k_\ell$.
- $Y.\text{weight}$, which equals $\sum_{\ell \in Y} k_\ell$. This field is needed for technical reasons, to be explained later.

For the purpose of analysis, we assume that the $\text{exposed}$ fields are represented as balanced binary search trees (BSTs), which ensures $O(\log k) = O(\log M \mathcal{P})$ time per access and update. The $\text{map}$ fields are also implemented using BSTs. We store the set $J_U = \{i \in [k] : U(i) \neq \emptyset\}$ as a BST, enabling is to determine in $O(\log k)$ time if an index $i$ is in $J_U$, and, if this is the case, to access $Y.\text{map}(i)$. The latter is also stored as a BST, allowing us to search and update $Y.\text{map}(i)$ in $O(\log |U(i)|) = O(\log M \mathcal{P})$ time. Note that, in practice, hashing may be a better alternative for both $\text{exposed}$ and $\text{map}$ fields, as it offers expected constant time performance per operation.

The data fields listed above allow us to efficiently retrieve the set $S$ of semi-universal labels in $U$, as needed in line 5 of $\text{BuildNTN}(\mathcal{P})$. Indeed, suppose that $U$ is the valid position extracted from $Q$ at the beginning of an iteration of the while loop of Lines 3–14, and that $Y = V(H_\mathcal{P}(U))$. Then, by Lemma 5, we have $S = \{v \in Y.\text{map}(i) : i \in Y.\text{exposed}\}$.

What remains is to devise an efficient way to update these fields for each of the connected components of $H_\mathcal{P}(U)$ created by replacing $U$ with its successor in Line 11.

Let $U'$ be the value of $U$ after Line 11; thus, $U'$ is the successor of $U$. By Observation 2, $H_\mathcal{P}(U')$ is obtained from $H_\mathcal{P}(U)$ through edge and node deletions. We need to

(a) Generate the new connected components resulting from these deletions, and
(b) Produce the required $\text{map}$, $\text{count}$, and $\text{exposed}$ data fields for the various connected components.

We accomplish (a) using the dynamic graph connectivity data structure of Holm et al. [20], which we refer to 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 \( N \) nodes, the amortized cost of each update is \( O(\log^2 N) \). Since \( H_P \) has \( O(M_P) \) nodes, each update takes \( O(\log^2 M_P) \) time. The total number of edge and node deletions performed by \( \text{BuildNTN}(P) \) —including all deletions in the iterations—is at most the total number of edges and nodes in \( H_P \), which is \( O(M_P) \). HDT allows us to maintain connectivity information throughout the entire algorithm in \( O(M_P \log^2 M_P) \) time, which is within the time bound claimed in Theorem 3.

For part (b), we need to augment HDT in order to maintain the required data fields for the various connected components created during edge and node deletion. In the next subsections, we describe how to do this. We begin by explaining how to initialize all the required data fields for \( H_P = \text{BuildNTN}(U_{\text{root}}) \).

### Initializing the data fields

Graph \( H_P(U_{\text{root}}) \) has a single connected component, \( Y_{\text{root}} = L(P) \), which is the entire vertex set of the graph. We initialize the required data fields as follows.

- For each \( i \in [k] \), \( Y_{\text{root}}. \text{map}(i) = \{r(T_i)\} \). This takes \( O(k) \) time.
- \( Y_{\text{root}}. \text{weight} = \sum_{\ell \in L(P)} k_{\ell} \). This takes \( O(M_P) \) time.

We initialize the \text{count} fields in \( O(M_P) \) time as follows:

1. Set \( \ell. \text{count} = 0 \) for all \( \ell \in L(P) \).
2. For each \( i \in [k] \) do the following.
   
   - (a) Let \( \rho_i \) denote the label of \( r(T_i) \).
   - (b) Increment \( \rho_i. \text{count} \) by one.

Once the \text{count} fields are initialized, it is easy to initialize \( Y_{\text{root}}. \text{exposed} \) in \( O(k) \) time. Thus, we can initialize all the required fields in \( O(M_P) \) time.

### Maintaining the data fields

Suppose that all data fields are correctly computed for every connected component that is in \( Q \) at the beginning of an iteration of the \text{while} loop in 3–14 of \( \text{BuildNTN} \). We now show how to generate the same fields efficiently for the new connected components created by Line 11.

### Computing successor positions

Let \( U \) be the valid position extracted from \( Q \) at the beginning of an iteration of \( \text{BuildNTN} \)’s \text{while} loop, and let \( Y = V(\text{Desc}(U)) \) be the associated connected component. Assume all the data fields for \( Y \) have been correctly computed. To obtain the successor of \( U \) in Line 11 of \( \text{BuildNTN} \), we perform the following steps.

1. Identify the set \( S \) of semi-universal labels in \( U \). As we saw, this set is given by \( S = \{\ell \in Y. \text{map}(i) \mid i \in Y. \text{exposed}\} \).
2. Set \( Y. \text{map}(i) = \emptyset \) for every \( i \in Y. \text{exposed} \).
3. Make \( Y. \text{exposed} = \emptyset \).
4. For each \( \ell \in S \) and each \( i \) such that \( \ell \in L(T_i) \), do the following.
   - If \( \text{Ch}_1(\ell) \neq \emptyset \), replace \( Y. \text{map}(i) \) by \( \text{Ch}_1(\ell) \). If \( \text{Ch}_1(\ell) \) is a singleton set \( \{\alpha\} \), increment \( \alpha. \text{count} \) by one. If \( \alpha. \text{count} = k_\ell \), add \( i \) to \( Y. \text{exposed} \).
   - Otherwise, \( Y. \text{map}(i) \) is undefined.
5. For each label \( \ell \) in \( S \), delete the edges incident on \( \ell \) and then \( \ell \) itself, updating the data fields as necessary after each deletion.

The total number of operations on \text{map} and \text{exposed} fields in Steps 1–4 is \( O(\sum_{\ell \in S} k_\ell) \). Since each label becomes semi-universal at most once, the total number of operations on \text{map} fields over the entire execution of \( \text{BuildNTN}(P) \) is \( O(\sum_{\ell \in L(P)} k_\ell) \), which is \( O(M_P) \). The same bound holds for updates to \text{count} and \text{exposed} fields.

Next let us focus on how to handle the deletion of a single edge in Step 5.

### Deleting an edge

To delete an edge between \( \ell \) and a child \( \alpha \) of \( \ell \), we proceed as follows.

1. Delete \((\ell, \alpha)\), querying HDT to determine whether this disconnects \( Y \).
   - If \( Y \) remains connected, skip the next steps and proceed directly to the next child of \( \ell \).
   - Otherwise, \( Y \) is split into two components, \( Y_1 \) and \( Y_2 \).
2. Update \( Y_1. \text{weight} \) and \( Y_2. \text{weight} \).
3. Identify which of \( Y_1 \) and \( Y_2 \) has the smaller weight field. Without loss of generality, assume that \( Y_1. \text{weight} \leq Y_2. \text{weight} \).
4. Initialize \( Y_1. \text{map} \) and \( Y_1. \text{exposed} \) to \text{null}.
5. Initialize \( Y_2. \text{map} \) and \( Y_2. \text{exposed} \) to \( Y. \text{map} \) and \( Y. \text{exposed} \), respectively.
6. For each label \( \beta \) in \( Y_1 \), perform the following steps for each \( i \) such that \( \beta \in L(T_i) \).

   - (a) Delete \( \beta \) from \( Y_2. \text{map}(i) \) and add \( \beta \) to \( Y_1. \text{map}(i) \).
   - (b) Adjust \text{count} and \text{exposed} fields as necessary.

The connectivity test in Step 1 is done by querying HDT. Steps 3–5 are trivial. We thus focus on Steps 2 and 6.
To perform Step 2, we use the well-known technique of scanning the smaller component [21]. We first consult HDT to determine which of $Y_1$ or $Y_2$ has fewer nodes. Assume, without loss of generality, that $|Y_1| \leq |Y_2|$. We initialize $Y_1.weight$ to 0 and $Y_2.weight$ to $Y.weight$. We then scan the labels of $Y_1$, incrementing $Y_1.weight$ by $k_ℓ$ for each label $ℓ \in Y_1$. When the scan of $Y_1$ is complete, we make $Y_1.weight = Y_2.weight = Y.weight$. We claim that any label $ℓ \in L(\mathcal{P})$ is scanned $O(\log M_P)$ times over the entire execution of BuildNTN($\mathcal{P}$). To verify this, let $N(ℓ)$ be the number of nodes in the connected component containing $ℓ$. Suppose that, initially, $N(ℓ) = N$. Then, the $r$th time we scan $ℓ$, $N(ℓ) \leq N/2^r$. Thus, $ℓ$ is scanned $O(\log N)$ times. The claim follows, since $N = O(M_P)$. Therefore, the total number of updates over all labels is $O(M_P \log M_P)$.

Each execution of Step 6(a) updates each of $Y_1.map(i)$ and $Y_2.map(i)$ once. Step 6(b) is more complex, but can also be accomplished with $O(1)$ data field updates. We omit the (tedious) details. In summary, each execution of step 6 for some $β \in L(\mathcal{P})$ performs $O(k_β)$ data field updates.

Let us track the number of data field updates in Step 6 that can be attributed to some specific label $β \in L(\mathcal{P})$ over the entire execution of BuildNTN($\mathcal{P}$). Let $w_r(β)$ be the weight of the connected component containing $β$ at the beginning of Step 6, on the $r$th time that $β$ is considered in that step. Thus, $w_0(β) \leq \sum_{ℓ \in L(\mathcal{P})} k_ℓ$. We claim that $w_r(β) \leq w_0(β)/2^r$. The reason is that we only consider $β$ if (a) $β$ is contained in one of the two components that result from deleting an edge in step 1 and (b) the component containing $β$ has the smaller weight. Hence, the number of times $β$ is considered in step 6 over the entire execution of BuildNTN($\mathcal{P}$) is $O(\log w_0(β))$, which is $O(\log M_P)$, since $w_0(β) = O(M_P)$. Therefore, the total number of data field updates in Step 6, over all labels in $L(\mathcal{P})$ considered throughout the entire execution of BuildNTN($\mathcal{P}$), is $O(\log M_P \cdot \sum_{ℓ \in L(\mathcal{P})} k_ℓ)$, which is $O(M_P \log M_P)$.

We conclude that the total running time of BuildNTN($\mathcal{P}$) is $O(M_P \log^2 M_P)$, completing the proof of Theorem 3.

Discussion

Like our earlier algorithm for compatibility of ordinary phylogenetic trees, the more general algorithm presented here, BuildNTN, is a polylogarithmic factor away from optimality (a trivial lower bound is $Ω(M_P)$, the time to read the input). BuildNTN has a linear-space implementation, using the results of Thorup [22]. A question to be investigated next is the performance of the algorithm on real data. Another important issue is integrating our algorithm into a synthesis method that deals with incompatible profiles.

Authors’ contributions

Algorithms and proofs were developed jointly by YD and DFB. The first draft of the paper was written by DFB, with substantial contributions from YD. YD and DFB both proofread the manuscript. Both authors read and approved the final manuscript.

Acknowledgements

The authors were supported in part by the National Science Foundation under Grant CCF-1422134.

Competing interests

The authors declare that they have no competing interests.

Received: 23 December 2016 Accepted: 4 March 2017 Published online: 16 March 2017

References

1. Aho AV, Sagiv Y, Szymanski TG, Ullman JD. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. SIAM J Comput. 1981;10(3):405–21.
2. Deng Y, Fernández-Baca D. Fast compatibility testing for rooted phylogenetic trees. In: Grossi R, Lewenstein M (editors) 27th annual symposium on combinatorial pattern matching (CPM 2016). Leibniz International Proceedings in Informatics. Schloss Dagstuhl—Leibniz-Zentrum für Informatik. Dagstuhl Publishing; 2016. p. 12. doi:10.4230/LIPIcs.CPM.2016.12.
3. Henzinger MR, King V, Warnow T. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. Algorithmica. 1999;24:1–13.
4. Steel MA. The complexity of reconstructing trees from qualitative characters and subtrees. J Classif. 1992;9:91–116.
5. Baum BR. Combining trees as a way of combining data sets for phylogenetic inference, and the desirability of combining gene trees. Taxon. 1992;41:3–10.
6. Ragan MA. Phylogenetic inference based on matrix representation of trees. Mol Phylogenet Evol. 1992;1:53–8.
7. Bininda-Emonds ORP, Cardillo M, Jones KE, MacPhee RDE, Beck RMD, Greiner R, Price SA, Vos RA, Gittleman JL, Purvis A. The delayed rise of present-day mammals. Nature. 2007;446:507–12.
8. Page RM. Taxonomy, supertrees, and the tree of life. In: Bininda-Emonds ORP, editor. Phylogenetic supertrees: combining information to reveal the tree of life. Dordrecht: Kluwer; 2004. p. 247–65.
9. Sanderson MJ. Phylogenetic signal in the eukaryotic tree of life. Science. 2008;321(5885):121–3.
10. Sayers EW, et al. Database resources of the national center for biotechnology information. Nucl Acids Res. 2009;37(Database issue):5–15.
11. The Angiosperm Phylogeny Group. An update of the Angiosperm Phylogeny Group classification for the orders and families of flowering plants: APG IV. Bot J Linn Soc. 2016;181:1–20.
12. Hinchliff CE, Smith SA, Allman JF, Burleigh JG, Chaudhary R, Coghill LM, Crandall KA, Deng J, Drew BT, Gazis R, Gude K, Hibbett DS, Katz LA, Laughinghouse HD IV, McTavish EJ, Midford PE, Owien CL, Reed RH, Reesk JA, Solis DE, Williams T, Cranston KA. Synthesis of phylogeny and taxonomy into a comprehensive tree of life. Proc Natl Acad Sci. 2015;112(41):12764–9. doi:10.1073/pnas.1423041112.
13. Bordewich M, Evans G, Semple C. Extending the limits of supertree methods. Ann Comb. 2006;10:31–51.

Submit your next manuscript to BioMed Central and we will help you at every step:

- We accept pre-submission inquiries
- Our selector tool helps you to find the most relevant journal
- We provide round the clock customer support
- Convenient online submission
- Thorough peer review
- Inclusion in PubMed and all major indexing services
- Maximum visibility for your research

Submit your manuscript at www.biomedcentral.com/submit