Deng and Fernández-Baca *Algorithms Mol Biol (2017) 12:7*
DOI 10.1186/s13015-017-0099-7

Algorithms for Molecular Biology

RESEARCH Open Access

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 $\mathcal{P}$ 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 $\mathcal{P}$. 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 $\mathcal{P}$.

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

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 $\mathcal{P}$. 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 $\mathcal{P} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k\}$ of rooted phylogenetic trees with partially overlapping taxon sets. $\mathcal{P}$ is called a *profile* and the trees in $\mathcal{P}$ are the *input trees*. The question is whether there exists a tree $\mathcal{T}$ whose taxon set is the union of the taxon sets of the input trees, such that $\mathcal{T}$ 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 $\mathcal{T}$. 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 $\mathcal{T}$ whose taxon set is the union of the taxon sets of the input trees, such that $\mathcal{T}$ exhibits not only the clusterings among the taxa, but also the ancestor/descendant relationships among taxa in the input trees. Our main result is a $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ algorithm for the compatibility problem for trees with nested taxa, where $M_{\mathcal{P}}$ is the total number of nodes and edges in the trees in $\mathcal{P}$.

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

*Correspondence: fernande@iastate.edu
[†]Yun Deng and David Fernández-Baca contributed equally
Department of Computer Science, Iowa State University, Atanasoff Hall, Ames, IA, USA

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 structure and completeness that might be hard to obtain otherwise. A recent example of the utility of taxonomies is the Open Tree of Life, a draft phylogeny for over 2.3 million species [12].

Taxonomies are not, strictly speaking, phylogenies. In particular, their internal nodes and some of their leaves are labeled with higher-order taxa. Nevertheless, taxonomies have many of the same mathematical characteristics as phylogenies. Indeed, both phylogenies and taxonomies are *semi-labeled trees* [13, 14]. We will use this term throughout the rest of the paper to refer to trees with nested taxa.

The fastest previous algorithm for testing ancestral compatibility, based on earlier work by Daniel and Semple [15], is due to Berry and Semple [16]. Their algorithm runs in $O\left(\log^2 n \cdot \tau_{\mathcal{P}}\right)$ time using $O(\tau_{\mathcal{P}})$ space. Here, n is the number of distinct taxa in $\mathcal{P}$ and $\tau_{\mathcal{P}} = \sum_{i=1}^{k} \sum_{v \in I(\mathcal{T}_i)} d(v)^2$, where $I(\mathcal{T}_i)$ is the set of internal nodes of $\mathcal{T}_i$, for each $i \in \{1, \ldots, k\}$, and $d(v)$ is the degree of node v. While the algorithm is polynomial, its dependence on node degrees is problematic: semi-labeled trees can be highly unresolved (i.e., contain nodes of high degree), especially if they are taxonomies.

Our contributions

As stated earlier, our main result is an algorithm to test ancestral compatibility that runs in $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time, using $O(M_{\mathcal{P}})$ space. These bounds are independent of the degrees of the nodes of the input trees, a valuable characteristic for large datasets that include taxonomies. To achieve our result, we extend ideas from our recent algorithm for testing the compatibility of ordinary phylogenetic trees [2]. As in that algorithm, a central notion in the current paper is the *display graph* of profile $\mathcal{P}$, denoted $H_{\mathcal{P}}$. This is the graph obtained from the disjoint union of the trees in $\mathcal{P}$ by identifying nodes that have the same label (see the section titled "Testing ancestral compatibility"). The term "display graph" was introduced by Bryant and Lagergren [17], but similar ideas have been used elsewhere. In particular, the display graph is closely related to Berry and Semple's *restricted descendancy graph* [16], a mixed graph whose directed edges correspond to the (undirected) edges of $H_{\mathcal{P}}$ and whose undirected edges have no correspondence in $H_{\mathcal{P}}$. The second kind of edges are the major component of the $\tau_{\mathcal{P}}$ term in the time and space complexity of Berry and Semple's algorithm. The absence of such edges makes $H_{\mathcal{P}}$ significantly smaller than the restricted descendancy graph. Display graphs also bear some relation to *tree alignment graphs* [18].

Here, we exploit the display graph more extensively than in our previous work. Although the display graph of 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, BuildNT$_{\mathrm{N}}$, that is easier to implement. We also give an example of the execution of BuildNT$_{\mathrm{N}}$. The "Implementation" section gives the implementation details for BuildNT$_{\mathrm{N}}$. 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 $u \preceq_T v$, if u lies on the path from v to $r(T)$ in T. If $u \preceq_T v$, then v is a *descendant* of u. Node u is a *proper descendant* of v if u is a descendant of v and $v \neq u$. If $\{u, v\} \in E(T)$ and $u \preceq_T v$, then u is the *parent* of v and v is a *child* of u. If neither $u \preceq_T v$ nor $v \preceq_T u$ 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 $\mathcal{T} = (T, \phi)$ where T is a tree and ϕ is a mapping from a set $L(\mathcal{T})$ to $V(T)$ such that, for every node $v \in V(T)$ of degree at most two,

$v \in \phi(L(\mathcal{T}))$. $L(\mathcal{T})$ is the *label set* of $\mathcal{T}$ and ϕ is the *labeling function* of $\mathcal{T}$.

For every node $v \in V(T)$, $\phi^{-1}(v)$ denotes the (possibly empty) subset of $L(\mathcal{T})$ 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, including 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 $\mathcal{T} = (T, \phi)$ is *singularly labeled* if every node in T has at most one label; $\mathcal{T}$ 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 $\mathcal{T} = (T, \phi)$ where $r(T)$ has degree at least two and ϕ is a bijection from $L(\mathcal{T})$ into leaf set of T (thus, internal nodes are not labeled).

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

Two semi-labelled trees $\mathcal{T} = (T, \phi)$ and $\mathcal{T}' = (T', \phi')$ are *isomorphic* if there exists a bijection $\psi : V(T) \to V(T')$ such that $\phi' = \psi \circ \phi$ and, for any two nodes $u, v \in V(T)$, $(u, v) \in E(T)$ if and only $(\psi(u), \psi(v)) \in E(T')$.

Let $\mathcal{T} = (T, \phi)$ 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, $X(u) = \bigcup_{v: u \leq_T v} \phi^{-1}(v)$. $X(u)$ is called a *cluster* of T. $\mathrm{Cl}(\mathcal{T})$ denotes the set of all clusters of $\mathcal{T}$. It is well known [14, Theorem 3.5.2] that a semi-labeled tree $\mathcal{T}$ is completely determined by $\mathrm{Cl}(\mathcal{T})$. That is, if $\mathrm{Cl}(\mathcal{T}) = \mathrm{Cl}(\mathcal{T}')$ for some other semi-labeled tree $\mathcal{T}'$, then $\mathcal{T}$ is isomorphic to $\mathcal{T}'$.

Suppose $A \subseteq L(\mathcal{T})$ for a semi-labeled tree $\mathcal{T} = (T, \phi)$. The *restriction* of $\mathcal{T}$ to A, denoted $\mathcal{T}|A$, is the semi-labeled tree whose cluster set is $\mathrm{Cl}(\mathcal{T}|A) = \{X \cap A : X \in \mathrm{Cl}(\mathcal{T}) \text{ and } X \cap A \neq \emptyset\}$. Intuitively, $\mathcal{T}|A$ 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 $\mathcal{T} = (T, \phi)$ and $\mathcal{T}' = (T', \phi')$ be semi-labeled trees such that $L(\mathcal{T}') \subseteq L(\mathcal{T})$. $\mathcal{T}$ *ancestrally displays* $\mathcal{T}'$ if $\mathrm{Cl}(\mathcal{T}') \subseteq \mathrm{Cl}(\mathcal{T}|L(\mathcal{T}'))$. Equivalently, $\mathcal{T}$ ancestrally displays $\mathcal{T}'$ if $\mathcal{T}'$ can be obtained from $\mathcal{T}|L(\mathcal{T}')$ by contracting edges, and, for any $\ell_1, \ell_2 \in L(\mathcal{T}')$,

(i) if $\ell_1 <_{\mathcal{T}'} \ell_2$, then $\ell_1 <_{\mathcal{T}} \ell_2$, and
(ii) if $\ell_1 \parallel_{\mathcal{T}'} \ell_2$, then $\ell_1 \parallel_{\mathcal{T}} \ell_2$.

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 $\mathcal{T}$, let us define $D(\mathcal{T})$ and $N(\mathcal{T})$ as follows.

$$D(\mathcal{T}) = \{(\ell, \ell') : \ell, \ell' \in L(\mathcal{T}) \text{ and } \ell <_{\mathcal{T}} \ell'\}$$
$$N(\mathcal{T}) = \{\{\ell, \ell'\} : \ell, \ell' \in L(\mathcal{T}) \text{ and } \ell \parallel_{\mathcal{T}} \ell'\}$$

Note that $D(\mathcal{T})$ consists of *ordered* pairs, while $N(\mathcal{T})$ consists of *unordered* pairs.

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

Profiles and ancestral compatibility

Throughout the rest of this paper $\mathcal{P} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k\}$ denotes a set where, for each $i \in [k]$, $\mathcal{T}_i = (T_i, \phi_i)$ is a semi-labeled tree. We refer to $\mathcal{P}$ as a *profile*, and write $L(\mathcal{P})$ to denote $\bigcup_{i \in [k]} L(\mathcal{T}_i)$, the *label set* of $\mathcal{P}$. Figure 1 shows a profile where $L(\mathcal{P}) = \{a, b, c, d, e, f, g, h, i\}$. 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 $M_{\mathcal{P}} = |V(\mathcal{P})| + |E(\mathcal{P})|$.

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

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

$$\mathcal{P}|X = \{\mathcal{T}_1|X \cap L(\mathcal{T}_1), \mathcal{T}_2|X \cap L(\mathcal{T}_2), \ldots, \mathcal{T}_k|X \cap L(\mathcal{T}_k)\}.$$

The proof of the following lemma is straightforward.

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

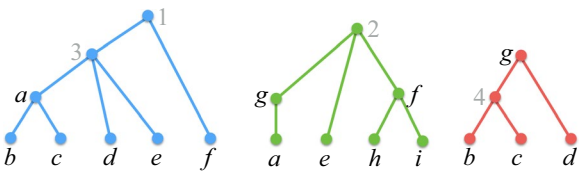

Fig. 1 A profile $\mathcal{P} = \{\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3\}$—trees are ordered left-to-right. The *letters* are the original labels; *grey numbers* are labels added to make the trees fully labeled (Adapted from [16])

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 $\phi_i^{-1}(v) = \{\ell\}$, 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 $i \in [k]$, $\mathcal{T}_i$ 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 $U(i) \subseteq L(\mathcal{T}_i)$, 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 $\mathrm{Desc}_i(U) = \{\ell : \ell' \preceq_{\mathcal{T}_i} \ell$, for some $\ell' \in U(i)\}$, and let $\mathrm{Desc}_\mathcal{P}(U) = \bigcup_{i\in[k]} \mathrm{Desc}_i(U)$.

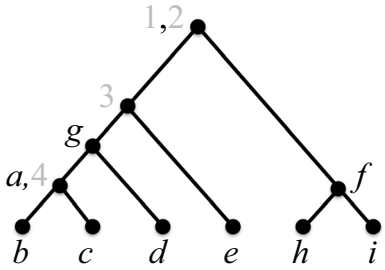

Fig. 2 A tree $\mathcal{T}$ that ancestrally displays the profile of Fig. 1 (Adapted from [16])

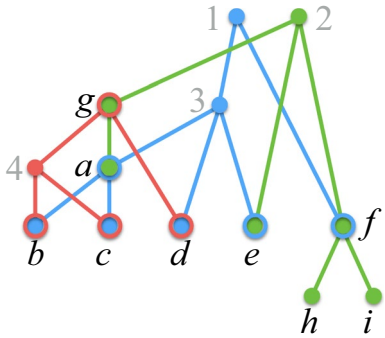

Fig. 3 The display graph $H_\mathcal{P}$ for the profile of Fig. 1

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 $\mathcal{T}_i$ and
(V2) $\mathrm{Desc}_i(U) = \mathrm{Desc}_\mathcal{P}(U) \cap L(\mathcal{T}_i)$.

Lemma 4 *For any valid position U,*

$$\mathcal{P}|\mathrm{Desc}_\mathcal{P}(U) = \{\mathcal{T}_1|\mathrm{Desc}_1(U), \mathcal{T}_2|\mathrm{Desc}_1(U), \ldots, \mathcal{T}_k|\mathrm{Desc}_k(U)\}.$$

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

For any valid position U, $H_\mathcal{P}(U)$ denotes the subgraph of $H_\mathcal{P}$ induced by $\mathrm{Desc}_\mathcal{P}(U)$.

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

A valid position of special interest to us is U_{root}, the *root position*, defined as follows.

$$U_{\mathrm{root}} = (\phi_i^{-1}(r(T_1)), \phi_i^{-1}(r(T_2)), \ldots, \phi_i^{-1}(r(T_k))). \quad (1)$$

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

Semi-universal labels

Let U be a valid position, and let ℓ be a label in U. Then, ℓ is *semi-universal in U* if $U(i) = \{\ell\}$, for every $i \in [k]$ such that $\ell \in L(\mathcal{T}_i)$. In Fig. 3, labels 1 and 2 are semi-universal in U_{root}, but g is not, since g is in both $L(\mathcal{T}_2)$ and $L(\mathcal{T}_3)$, but $U_{\mathrm{root}}(2) \neq \{g\}$.

The term "semi-universal", borrowed from Pe'er et al. [19], derives from the following fact. Suppose that $\mathcal{P}$ is ancestrally compatible, that $\mathcal{T}$ is a tree that ancestrally displays $\mathcal{P}$, and that ℓ is a semi-universal label for some valid position U. Then, as we shall see, ℓ must label the root u_ℓ of a subtree of $\mathcal{T}$ that contains all the descendants of ℓ in $\mathcal{T}_i$, for every i such that $\ell \in L(\mathcal{T}_i)$. The qualifier "semi" is because this subtree may also contain labels that do not descend from ℓ in any input tree, but descend instead from some other semi-universal label ℓ' in U. In this case, ℓ' also labels u_ℓ. 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_ℓ denote the number of input trees that contain label ℓ. We can obtain k_ℓ 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 ℓ is semi-universal in U if the cardinality of the set $J_\ell = \{i \in [k] : U(i) = \{\ell\}\}$ equals k_ℓ.*

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

Successor positions

For every $i \in [k]$ and every $\ell \in L(\mathcal{T}_i)$, let $\mathrm{Ch}_i(\ell)$ denote the set of children of ℓ in $L(\mathcal{T}_i)$. For a subset A of $L(\mathcal{T}_i)$, let $\mathrm{Ch}_i(A) = \bigcup_{\ell \in A} \mathrm{Ch}_i(\ell)$. 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 $U' = (U'(1), U'(2), \ldots, U'(k))$, where, for each $i \in [k]$, $U'(i)$ is defined as follows.

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

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

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 ℓ and $\mathrm{Ch}_i(\ell)$; (2) delete ℓ.*

Let U be a valid position, and W be a subset of $\mathrm{Desc}_{\mathcal{P}}(U)$. 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_{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 (V1) 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 (V1), since U does. Suppose $\ell \in S$. Then, for each $i \in [k]$ such that $\ell \in L(\mathcal{T}_i)$, $\mathrm{Desc}_i(U') = \mathrm{Desc}_i(U) \setminus \{\ell\}$ and $\mathrm{Desc}_{\mathcal{P}}(U') \cap L(\mathcal{T}_i) = (\mathrm{Desc}_{\mathcal{P}}(U) \cap L(\mathcal{T}_i)) \setminus \{\ell\}$. Thus, since (V2) holds for U, it also holds for U'. □

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 $\mathcal{P}|\mathrm{Desc}_{\mathcal{P}}(U)$ 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 subprofile is compatible. If all the recursive calls are successful, then their results are combined into a supertree for $\mathcal{P}|\mathrm{Desc}_{\mathcal{P}}(U)$.

In detail, BuildNT proceeds as follows. Line 1 computes the set S of semi-universal labels in U. If S is empty, then, $\mathcal{P}|\mathrm{Desc}_{\mathcal{P}}(U)$ is incompatible, and, thus, so is $\mathcal{P}$. This fact is reported in Line 3. Line 4 creates a tentative root r_U, labeled by S, for the tree $\mathcal{T}_U$ for $L(U)$. Line 5 checks if S contains exactly one label ℓ, with no proper descendants. If so, by the connectivity assumption, ℓ must be the sole member of $\mathrm{Desc}_{\mathcal{P}}(U)$; 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 with respect to S. Let $W_1, W_2, \ldots, W_p$ be the connected components of $H_{\mathcal{P}}(U)$ after updating U. 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 $\mathcal{P}|\mathrm{Desc}_{\mathcal{P}}(U \cap W_j)$. If any subproblem is incompatible, Line 12 reports that $\mathcal{P}$ is incompatible. Otherwise, Line 13 returns the tree obtained by making the t_js the subtrees of root r_U.

Algorithm 1: BUILDNT(U)

Input: A valid position U for $\mathcal{P}$ such that $H_{\mathcal{P}}(U)$ is connected.
Output: A semi-labeled tree that ancestrally displays $\mathcal{P}' = \mathcal{P}|\mathrm{Desc}_{\mathcal{P}}(U)$, if $\mathcal{P}'$ is ancestrally compatible; `incompatible` otherwise.

```
1  Let S = {ℓ ∈ U : ℓ is semi-universal in U}
2  if S = ∅ then
3  |  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  |  return r_U
7  Replace U by the successor of U with respect to S
8  Let W_1, W_2, ..., W_p be the connected components of H_P(U)
9  foreach j ∈ [p] do
10 |  Let t_j = BUILDNT(U|W_j)
11 |  if t_j is not a tree then
12 |  |  return incompatible
13 return the tree with root r_U and subtrees t_1, ..., t_p
```

Next, we argue the correctness of BuildNT.

Correctness

Lemma 7 *Let U be a valid position in $\mathcal{P}$. If* BuildNT(U) *returns a tree $\mathcal{T}_U$, then $\mathcal{T}_U$ is a phylogenetic tree such that $L(\mathcal{T}_U) = L(U)$.*

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 $H_{\mathcal{P}}(U)$ in step 8. Since BuildNT(U) returns tree $\mathcal{T}_U$, 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 $j \in [p]$. 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 $L(U)$. □

Theorem 1 *Let $\mathcal{P} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k\}$ be a profile and let U_{root} be the root position, as defined in Eq. (1). Then,* BuildNT(U_{root}) *returns either (i) a semi-labeled tree $\mathcal{T}$ that ancestrally displays $\mathcal{P}$, if $\mathcal{P}$ is ancestrally compatible, or (ii)* `incompatible` *otherwise.*

Proof BuildNT(U_{root}) either returns a tree or `incompatible`. We consider each case separately.

(i) Suppose that BuildNT(U_{root}) returns a semi-labeled tree $\mathcal{T}$. By Lemma 7, $L(\mathcal{T}) = L(\mathcal{P})$. We prove that $\mathcal{T}$ ancestrally displays $\mathcal{P}$. By Lemma 1, it suffices to show that $D(\mathcal{T}_i) \subseteq D(\mathcal{T})$ and $N(\mathcal{T}_i) \subseteq N(\mathcal{T})$, for each $i \in [k]$. Consider any $(\ell, \ell') \in D(\mathcal{T}_i)$. Then, ℓ has a child ℓ'' in $\mathcal{T}_i$ such that $\ell'' \preceq_{\mathcal{T}_i} \ell'$ —note that we may have $\ell'' = \ell$. There must be a recursive call to BuildNT(U), for some valid position U, where ℓ is the set S of semi-universal labels obtained in Line 1. By Observation 2, label ℓ'', and thus ℓ', both lie in one of the connected components of the graph obtained by deleting all labels in S, including ℓ, and their incident edges from $H_{\mathcal{P}}(U)$. It now follows from the construction of $\mathcal{T}$ that $(\ell, \ell') \in D(\mathcal{T})$. Thus, $D(\mathcal{T}_i) \subseteq D(\mathcal{T})$. Now, consider any $\{\ell, \ell'\} \in N(\mathcal{T}_i)$. Let ν be the lowest common ancestor of $\phi_i(\ell)$ and $\phi_i(\ell')$ in $\mathcal{T}_i$ and let ℓ_ν be the label of ν. Then, ℓ_ν has a pair of children, ℓ_1 and ℓ_2 say, in $\mathcal{T}_i$ such that $\ell_1 \preceq_{\mathcal{T}_i} \ell$, and $\ell_2 \preceq_{\mathcal{T}_i} \ell'$. Because BuildNT($U_{\mathrm{root}}$) returns a tree, there are recursive calls BuildNT(U_1) and BuildNT(U_2) for valid positions U_1 and U_2 such that ℓ_1 is semi-universal for U_1 and ℓ_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 ℓ_1 nor ℓ_2 is semi-universal, a contradiction. Further, it follows from the construction of $\mathcal{T}$ that we must have $\text{Desc}_{\mathcal{P}}(U_1) \cap \text{Desc}_{\mathcal{P}}(U_2) = \emptyset$. Hence, $\ell \parallel_{\mathcal{T}} \ell'$, and, therefore, $\{\ell, \ell'\} \in N(\mathcal{T})$.

(ii) Asssume, by way of contradiction, that BuildNT(U_{root}) returns *incompatible*, but that $\mathcal{P}$ is ancestrally compatible. By assumption, there exists a semi-labeled tree $\mathcal{T}$ that ancestrally displays $\mathcal{P}$. Since BuildNT(U_{root}) returns *incompatible*, there is a recursive call to 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, $\mathcal{T}|\text{Desc}_{\mathcal{P}}(U)$ ancestrally displays $\mathcal{P}|\text{Desc}_{\mathcal{P}}(U)$. Thus, by Lemma 4, $\mathcal{T}|\text{Desc}_{\mathcal{P}}(U)$ ancestrally displays $\mathcal{T}_i|\text{Desc}_i(U)$, for every $i \in [k]$. Let ℓ be any label in the label set of the root of $\mathcal{T}|\text{Desc}_{\mathcal{P}}(U)$. Then, for each $i \in [k]$ such that $\ell \in L(\mathcal{T}_i)$, ℓ must be the label of the root of $\mathcal{T}_i|\text{Desc}_i(U)$. Thus, for each such i, $U(i) = \{\ell\}$. Hence, ℓ is semi-universal in U, a contradiction.

□

$L(U)$ in the supertree. The body of the loop closely follows the steps performed by a call to BuildNT(U). Line 5 computes the set S of semi-universal labels in U. If S is empty, the algorithm reports that $\mathcal{P}$ is incompatible and terminates (Lines 6–7). The algorithm then creates a tentative root r_U labeled by S for the tree $\mathcal{T}_U$ 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 *while* loop, and *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 *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$_{\text{N}}$ processes connected components differs from that of BuildNT — breadth-first instead of depth-first—, it is straightforward

Algorithm 2: BUILDNT$_{\text{N}}(\mathcal{P})$

Input: A profile $\mathcal{P}$.
Output: A tree T that displays $\mathcal{P}$, if $\mathcal{P}$ is compatible; `incompatible` otherwise.

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

An iterative version

We now present BuildNT$_{\text{N}}$ (Algorithm 2), an iterative version of BuildNT, which lends itself naturally to an efficient implementation. BuildNT$_{\text{N}}$ 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, pred), where U is a valid position in $\mathcal{P}$ and pred is a reference to the parent of the node corresponding to U in the supertree built so far. BuildNT$_{\text{N}}$ simulates recursive calls in BuildNT by enqueuing pairs corresponding to subproblems. We explain this in more detail next.

BuildNT$_{\text{N}}$ initializes its queue to contain the starting position, U_{root}, with a null parent. It then proceeds to the *while* loop of Lines 3–14. Each iteration of the loop starts by dequeuing a valid position U, along with a reference pred to the potential parent for the subtree for

to see that the effect is equivalent, and the proof of correctness of BuildNT (Theorem 1) applies to BuildNT$_{\text{N}}$ as well. We thus state the following without proof.

Theorem 2 *Let $\mathcal{P} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k\}$ be a profile. Then,* BuildNT$_{\text{N}}(\mathcal{P})$ *returns either (i) a semi-labeled tree $\mathcal{T}$ that ancestrally displays $\mathcal{P}$, if $\mathcal{P}$ is ancestrally compatible, or (ii)* `incompatible` *otherwise.*

Let Q be BuildNT$_{\text{N}}$'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 pred. Let H_Q be the subgraph of $H_{\mathcal{P}}$ induced by $\bigcup\{\text{Desc}(U) : U \text{ is in } Q\}$. By Observation 1, H_Q is obtained from $H_{\mathcal{P}}$ through edge and node deletions.

Lemma 8 *At the start of any iteration of* BuildNT$_N$*'s while loop, the set of connected components of* H_Q *is* $\{V(H_{\mathcal{P}}(U)) : U \text{ is in } Q\}$.

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

In other words, Lemma 8 states that each iteration of BuildNT$_N(\mathcal{P})$ deals with a subgraph of $H_{\mathcal{P}}$, 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 BuildNT$_N$ on the profile $\mathcal{P} = (\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3)$ of Fig. 1. The figures show how the graph H_Q —initially equal to $H_{\mathcal{P}} = H_{\mathcal{P}}(U_{\text{root}})$ (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 BuildNT$_N(\mathcal{P})$ are denoted by U_α, for different subscripts α; S_α denotes the semi-universal labels in U_α, and U'_α denotes the successor of U_α with respect to S_α. We write L_α as an abbreviation for $L(U_\alpha)$ The root of the tree for L_α is r_{U_α} and is labeled by S_α.

Initially, $Q = ((U_{\text{root}}, \texttt{null}))$. 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_{root} is $S_{\text{root}} = \{1, 2\}$. Thus, $H_{\mathcal{P}}(U'_{\text{root}})$ 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,

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

After level 0 is processed, $Q = ((U_1, r_{U_{\text{root}}}), (U_2, r_{U_{\text{root}}}))$. Thus, 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 $S_1 = \{3\}$, so $H_{\mathcal{P}}(U'_1)$ has two components W_{11} and W_{12}. Let $U_{11} = U'_1|W_{11}$ and $U_{12} = U'_1|W_{12}$. Then,

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

We have $S_2 = \{f\}$, so $H_{\mathcal{P}}(U'_2)$ has two components W_{21} and W_{22}. Let $U_{21} = U'_2|W_{21}$ and $U_{22} = U'_2|W_{22}$. Then,

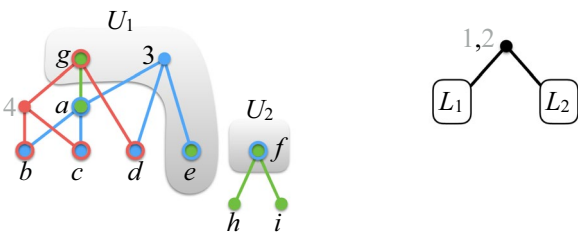

Fig. 4 After generating all supertree nodes in level 0

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

After level 1 is processed, $Q = ((U_{11}, r_1), (U_{12}, r_1), (U_{21}, r_2), (U_{22}, r_2))$.

Level 2. Refer to Fig. 6. We have $S_{11} = \{g\}$, so $H_{\mathcal{P}}(U'_{11})$ has two components W_{111} and W_{112}. Let $U_{111} = U'_{11}|W_{111}$ and $U_{112} = U'_{11}|W_{112}$. Then,

$$U_{111} = (\{a\}, \{a\}, \{4\}) \quad \text{and} \quad U_{112} = (\emptyset, \{d\}, \{d\}).$$

The only semi-universal labels in U_{12}, U_{21}, and U_{22} 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, $Q = ((U_{111}, r_{11}), (U_{112}, r_{11}))$.

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

$$U_{1111} = (\{b\}, \emptyset, \{b\}) \quad \text{and} \quad U_{1112} = (\{c\}, \emptyset, \{c\}).$$

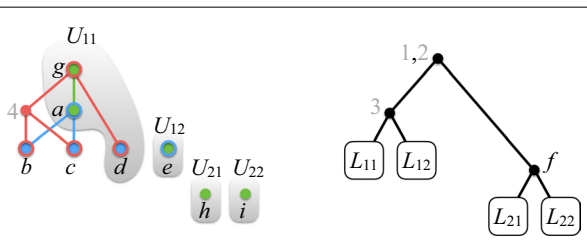

Fig. 5 After generating all supertree nodes in level 1

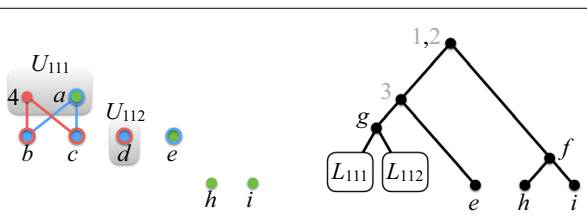

Fig. 6 After generating all supertree nodes in level 2

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, $Q = ((U_{1111}, r_{111}), (U_{1112}, r_{111}))$.

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 BuildNT$_N(\mathcal{P})$ 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 $O(M_\mathcal{P} \log^2 M_\mathcal{P})$ 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 BuildNT$_N$ so that the algorithm runs in $O(M_\mathcal{P} \log^2 M_\mathcal{P})$ 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 BuildNT$_N$'s queue. By Lemma 8, each valid position U in Q corresponds to one connected component of H_Q —namely Desc(U) —and vice-versa. We use this fact in the implementation of BuildNT$_N$: 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.

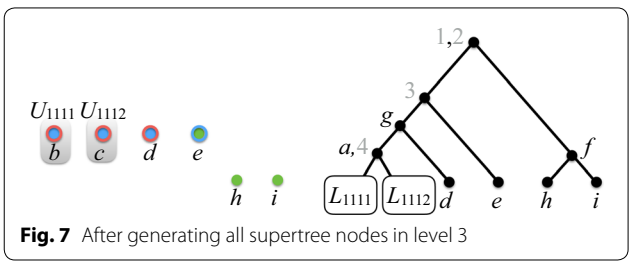

Fig. 7 After generating all supertree nodes in level 3

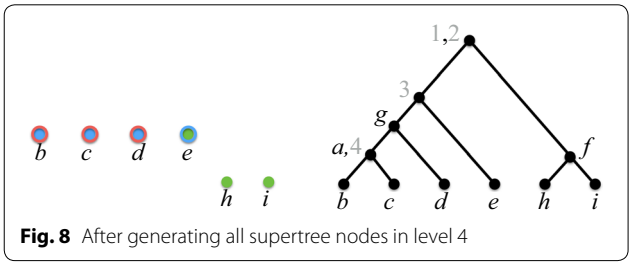

Fig. 8 After generating all supertree nodes in level 4

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 ℓ be any label in Y. Our implementation maintains the following data fields.

- Let $J_U = \{i \in [k] : U(i) \neq \emptyset\}$. Then, Y.`map` is a map from J_U to $L(U)$, where, for each $i \in J_U$, Y.`map`$(i) = U(i)$.
- For each $\ell \in Y$, ℓ.`count` equals the cardinality of the set $\{i \in [k] : Y$.`map`$(i) = \{\ell\}\}$. (Recall that k_ℓ is the number of input trees that contain ℓ.).
- Y.`exposed`, a set consisting of all $i \in [k]$ such that Y.`map`$(i) = \{\ell\}$ for some $\ell \in Y$ such that ℓ.`count` $= k_\ell$.
- Y.`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 `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 `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.`map`(i). The latter is also stored as a BST, allowing us to search and update Y.`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 `exposed` and `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 BuildNT$_N(\mathcal{P})$. 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 $S = \{v \in Y$.`map`$(i) : i \in Y$.`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 `map`, `count`, and `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 $H_{\mathcal{P}}$ has $O(M_{\mathcal{P}})$ nodes, each update takes $O(\log^2 M_{\mathcal{P}})$ time. The total number of edge and node deletions performed by BuildNT$_{\mathrm{N}}(\mathcal{P})$ —including all deletions in the interations—is at most the total number of edges and nodes in $H_{\mathcal{P}}$, which is $O(M_{\mathcal{P}})$. HDT allows us to maintain connectivity information throughout the entire algorithm in $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time, which is within the time bound claimed in Theorem 3.

For part (b), we need to augment HDT in order to maintain the 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 $H_{\mathcal{P}} = H_{\mathcal{P}}(U_{\mathrm{root}})$.

Initializing the data fields

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

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

We initialize the `count` fields in $O(M_{\mathcal{P}})$ time as follows:

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

Once the `count` fields are initialized, it is easy to initialize $Y_{\mathrm{root}}.\texttt{exposed}$ in $O(k)$ time. Thus, we can initialize all the required fields in $O(M_{\mathcal{P}})$ time.

Maintaining the data fields

Suppose that all data fields fields are correctly computed for every connected component that is in Q at the beginning of an iteration of the *while* loop in 3–14 of BuildNT$_{\mathrm{N}}$. 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 BuildNT$_{\mathrm{N}}$'s *while* loop, and let $Y = V(\mathrm{Desc}(U))$ 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 BuildNT$_{\mathrm{N}}$, we perform the following steps.

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

The total number of operations on `map` and `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 `map` fields over the entire execution of BuildNT$_{\mathrm{N}}(\mathcal{P})$ is $O(\sum_{\ell \in L(\mathcal{P})} k_\ell)$, which is $O(M_{\mathcal{P}})$. The same bound holds for updates to `count` and `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 ℓ and a child α of ℓ, we proceed as follows.

1. Delete (ℓ, α), querying HDT to determine whether this disconnects Y.
   - If Y remains connected, skip the next steps and proceed directly to the next child of ℓ.
   - Otherwise, Y is split into two components, Y_1 and Y_2.
2. Update $Y_1.\texttt{weight}$ and $Y_2.\texttt{weight}$.
3. Identify which of Y_1 and Y_2 has the smaller `weight` field. Without loss of generality, assume that $Y_1.\texttt{weight} \leq Y_2.\texttt{weight}$.
4. Initialize $Y_1.\texttt{map}$ and $Y_1.\texttt{exposed}$ to `null`.
5. Initialize $Y_2.\texttt{map}$ and $Y_2.\texttt{exposed}$ to $Y.\texttt{map}$ and $Y.\texttt{exposed}$, respectively.
6. For each label β in Y_1, perform the following steps for each i such that $\beta \in L(\mathcal{T}_i)$.
   (a) Delete β from $Y_2.\texttt{map}(i)$ and add β to $Y_1.\texttt{map}(i)$.
   (b) Adjust `count` and `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 $\ell \in Y_1$. When the scan of Y_1 is complete, we make Y_2.`weight` $= Y_2$.`weight` $- Y_1$.`weight`. We claim that any label $\ell \in L(\mathcal{P})$ is scanned $O(\log M_\mathcal{P})$ times over the entire execution of BuildNT$_\mathrm{N}(\mathcal{P})$. To verify this, let $N(\ell)$ be the number of nodes in the connected component containing ℓ. Suppose that, initially, $N(\ell) = N$. Then, the rth time we scan ℓ, $N(\ell) \leq N/2^r$. Thus, ℓ is scanned $O(\log N)$ times. The claim follows, since $N = O(M_\mathcal{P})$. Therefore, the total number of updates over all labels is $O(M_\mathcal{P} \log M_\mathcal{P})$.

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 $\beta \in L(\mathcal{P})$ performs $O(k_\beta)$ data field updates.

Let us track the number of data field updates in Step 6 that can be attributed to some specific label $\beta \in L(P)$ over the entire execution of BuildNT$_\mathrm{N}(\mathcal{P})$. Let $w_r(\beta)$ be the weight of the connected component containing β at the beginning of Step 6, on the rth time that β is considered in that step. Thus, $w_0(\beta) \leq \sum_{\ell \in L(\mathcal{P})} k_\ell$. We claim that $w_r(\beta) \leq w_0(\beta)/2^r$. 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 BuildNT$_\mathrm{N}(\mathcal{P})$ is $O(\log w_0(\beta))$, which is $O(\log M_\mathcal{P})$, since $w_0(\beta) = O(M_\mathcal{P})$. Therefore, the total number of data field updates in Step 6, over all labels in $L(\mathcal{P})$ considered throughout the entire execution of BuildNT$_\mathrm{N}(\mathcal{P})$, is $O(\log M_\mathcal{P} \cdot \sum_{\ell \in L(\mathcal{P})} k_\ell)$, which is $O(M_\mathcal{P} \log M_\mathcal{P})$.

Summary

Let us review the running times of each aspect of our implementation of BuildNT$_\mathrm{N}$.

- *Initializing the data structures*. This has two parts.
  - *Setting up the HDT data structure for $H_\mathcal{P}$*. This takes $O(M_\mathcal{P} \log^2 M_\mathcal{P})$ time.
  - *Initializing the data fields for the single connected component of $H_\mathcal{P}$*. This takes $O(M_\mathcal{P})$ time.
- *Maintaining the data structures*. This also has two parts.
  - *Updating the HDT data structure*. There are $O(M_\mathcal{P})$ edge and node deletions, at an amortized cost of $O(\log^2 M_\mathcal{P})$ per deletion, yielding a total time of $O(M_\mathcal{P} \log^2 M_\mathcal{P})$.
  - *Maintaining the relevant data fields for the connected components*. We have seen that the total number of updates is $O(M_\mathcal{P} \log M_\mathcal{P})$. Assume, conservatively, that each update can be done in $O(\log M_\mathcal{P})$ time. Then, this part takes a total of $O(M_\mathcal{P} \log^2 M_\mathcal{P})$ over the entire execution of BuildNT$_\mathrm{N}$.

We conclude that the total running time of BuildNT$_\mathrm{N}(\mathcal{P})$ is $O(M_\mathcal{P} \log^2 M_\mathcal{P})$, completing the proof of Theorem 3.

Discussion

Like our earlier algorithm for compatibility of ordinary phylogenetic trees, the more general algorithm presented here, BuildNT$_\mathrm{N}$, is a polylogarithmic factor away from optimality (a trivial lower bound is $\Omega(M_\mathcal{P})$, the time to read the input). BuildNT$_\mathrm{N}$ 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, Grenyer 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, Owen CL, Reed RH, Reesk JA, Soltis 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.
14. Semple C, Steel M. Phylogenetics. Oxford lecture series in mathematics. Oxford: Oxford University Press; 2003.
15. Daniel P, Semple C. Supertree algorithms for nested taxa. In: Bininda-Emonds ORP, editor. Phylogenetic supertrees: combining information to reveal the tree of life. Dordrecht: Kluwer; 2004. p. 151–71.
16. Berry V, Semple C. Fast computation of supertrees for compatible phylogenies with nested taxa. Syst Biol. 2006;55(2):270–88.
17. Bryant D, Lagergren J. Compatibility of unrooted phylogenetic trees is FPT. Theor Comput Sci. 2006;351:296–302.
18. Smith SA, Brown JW, Hinchliff CE. Analyzing and synthesizing phylogenies using tree alignment graphs. PLoS Comput Biol. 2013;9(9):1003223.
19. Pe'er I, Pupko T, Shamir R, Sharan R. Incomplete directed perfect phylogeny. SIAM J Comput. 2004;33(3):590–607. doi:10.1137/S0097539702406510.
20. Holm J, de Lichtenberg K, Thorup M. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. J ACM. 2001;48(4):723–60. doi:10.1145/502090.502095.
21. Even S, Shiloach Y. An on-line edge-deletion problem. J ACM. 1981;28(1):1–4. doi:10.1145/322234.322235.
22. Thorup M. Near-optimal fully-dynamic graph connectivity. In: Proceedings of the 32nd annual ACM symposium on theory of computing. ACM; 2000. p. 343–350.