Fast Algorithms for Inferring
Evolutionary Trees

TR 92-19
Richa Agarwala, David Fernandez-Baca, and Giora Slutzki

July 6, 1992

Iowa State University of Science and Technology
Department of Computer Science
226 Atanasoff
Ames, IA 50011
Fast Algorithms for Inferring Evolutionary Trees

RICH A AGARWALA, DAVID FERNÁNDEZ-BACA, AND GIORA SLUTZKI

Department of Computer Science, Iowa State University, Ames, IA 50011

July 3, 1992

Abstract

We present algorithms for the perfect phylogeny problem restricted to binary characters. The first algorithm is faster than a previous algorithm by Gusfield when the input matrix for the problem is sparse. Next, we present two online algorithms. For the first of these, the characters are given as input one at a time, while, for the second, the species are given as input one at a time. These two online algorithms can easily be combined into an algorithm that can process any sequence of additions and deletions of species and characters.

1 Introduction

A fundamental problem in molecular biology is that of inferring the evolutionary history of a set of species, each of which is specified by the set of traits or characters that it exhibits [3, 4]. Each character can occur in a species in one of a fixed number of states. Information about evolutionary history can be conveniently represented by an evolutionary or phylogenetic tree, often referred to simply as a phylogeny. More precisely, a phylogeny for a set of species \( S \) is a rooted tree in which the leaves represent the species in \( S \), and the internal nodes represent hypothetical ancestral species. The species are usually described by an \( n \times m \) matrix \( M \), where \( n = |S| \) is the number of species, \( m \) is the number of characters, and \( M_{ij} \) is the state of the \( j^{th} \) character for the \( i^{th} \) species. A perfect phylogeny \( P \), if it exists, is a phylogeny that assigns an \( m \)-vector of character states to each hypothesized ancestral species, and has the property that, for each state of each character, the set of nodes in \( P \) having that state is connected in \( P \). Recently, this problem was shown to be NP-complete by Bodlaender et. al. [1].

When we restrict the characters to have only two states, \( M \) becomes a 0-1 matrix and an alternative but equivalent representation of \( P \) is a rooted tree where each species in \( S \) is attached to exactly one leaf of \( P \), each character is associated with exactly one edge of \( P \), and for any leaf \( w \) of \( P \), the characters associated with the edges along the unique path from root to \( w \) exactly specify the characters of the species at leaf \( w \). See Figure 1 for an example. We shall refer to this special case of perfect phylogeny problem as the binary phylogeny (BP) problem. For the biological assumptions and interpretation of this problem, see [5].

Gusfield [5] gave a time-optimal algorithm for solving BP which takes time linear in the size of the input matrix \( M \). As he points out, the argument for the \( \Omega(nm) \) lower
bound considers a very unnatural matrix in which most of the species have most of the characters and $M$ has $nm - 2$ ones and only 2 zeroes. In general, however, $M$ is not dense. For such instances, the following representation of the input may be more natural: For each character $i \in \{1, \ldots, m\}$, we are given a set $C_i$ of all the species that exhibit that character; i.e., $C_i$ is the set of species which have a 1 in column $i$ of $M$. For convenience, we shall assume the existence of a universal set of species $C_0$ containing all $n$ species. From now on, we shall refer to $C_i$ as a character. We assume that all characters are nonempty. Our first result is a $O(C)$ algorithm for BP where $C = \sum_{i=1}^{m} |C_i|$. This is a significant improvement over the $O(nm)$ algorithm given in [5] for the case where $M$ is sparse. Next, we give two online algorithms. The first one receives characters as input one at a time. We suggest two implementations of this algorithm and show that depending on the implementation, the running time of the algorithm for adding a character $C_i$ is $O(|C_i| \log n)$ or $O(n)$. When invoked repeatedly for adding $m$ characters, this yields a total running time of $O(C \log n)$ or $O(nm)$. The second online algorithm receives species as input one at a time. Similar implementations as those of the previous algorithm result in running time of $O(|I(S_t)| \log m)$ or $O(m)$ where $S_t$ is the species being added and $|I(S_t)|$ is the number of ones in the row for $S_t$ in $M$. This leads to a total running time of $O(C \log m)$ or $O(nm)$ for adding $n$ species. Since, as pointed out in Sections 3 and 4, online deletion of characters and species from a given phylogeny can also be done quickly, we have a complete set of primitives for maintaining a phylogeny under any stream of additions and deletions of characters and species.

## 2 An algorithm for BP

Let $C = \{C_0, \ldots, C_m\}$ be a set of characters. Gusfield’s algorithm for BP and ours are based on the following fundamental result.

**Lemma 1** [2] $C$ has a perfect phylogenetic tree $P$ iff for every pair of characters $C_i, C_j$, either $C_i \cap C_j = \emptyset$ or one contains the other.
A set of characters \( \mathcal{C} \) satisfying the conditions of this lemma shall be said to be compatible.

Our approach is based on constructing a character tree (CT) \( T \) for \( \mathcal{C} \). \( T \) is a rooted tree with the following properties:

\begin{enumerate}
  \item[(C1)] \( V(T) = \{ N_0, \cdots, N_m \} \) where \( N_i \) is the node in \( T \) for character \( C_i \in \mathcal{C} \). \( N_0 \) is the root of \( T \).
  \item[(C2)] For every \( N_r \in V(T) \), if \( N_r \) has a parent \( N_p \), then \( C_r \subseteq C_p \).
  \item[(C3)] For every \( N_r \in V(T) \), \( C_r \cap C_s = \emptyset \), for each sibling \( N_s \) of \( N_r \).
\end{enumerate}

By the above lemma, it is clear that \( \mathcal{C} \) has a phylogenetic tree \( P \) iff it has a character tree. Given a character tree \( T \) for \( \mathcal{C} \), we can easily construct a phylogenetic tree \( P \) for \( \mathcal{C} \) as follows. First, for each \( r \in \{ 1, \cdots, m \} \), label the edge between \( N_i \) and \( N_p \) by \( r \), where \( N_p \) is the parent of \( N_i \). Next, consider each \( N_i \) in turn. Let \( N_i \) be the parent of \( N_i, \cdots, N_i \), and let \( C_R = \bigcup_{j=1}^{i} C_{i,j} \). If \( C_i \setminus C_R = \emptyset \), do nothing. Otherwise, proceed as follows. If \( N_i \) is a leaf and \( |C_i| = 1 \), label \( N_i \) with the single species \( x \in C_i \). Else, create a child of \( N_i \) for each \( x \in C_i \setminus C_R \) and label it by \( x \).

Figure 2 shows the characters and the character tree for the example in Figure 1. From now on, we concentrate on constructing \( T \). Algorithm \textsc{Phylogeny}, which is described below, creates \( T \) (if it exists) from the top down, inserting characters in nonincreasing order of cardinality. Each node has a list of links that point to the children of the node in \( T \). \textsc{Phylogeny} also uses an array \( A[1..n] \) where \( A[i] \) stores the name of the (unique)
lowest node in $T$ (constructed so far) containing species $i$. (Node $a$ is lower than node $b$
if the length of the path between $a$ and the root is greater than the length of the path
between $b$ and the root.)

**Algorithm Phylogeny**

**Step 1.** Sort $C_0, \ldots, C_m$ by nonincreasing cardinality. From now on, we assume without
loss of generality that $|C_0| \geq |C_1| \geq \cdots \geq |C_m|$. Create a node $N_i$ for each character
$C_i$ and initialize all $A[i]$’s to $N_0$.

**Step 2.**

\begin{algorithm}
\begin{algorithmic}
\For {$i = 1$ to $m$} 
\Begin
Pick a species $x \in C_i$; let $A[x] = N_i$
\For {every $y \in C_i$} 
\If {$A[y] \neq A[x]$} \textbf{return} FAILURE \Else $A[y] \leftarrow N_i$ \EndIf
\EndFor
\Add a link from $N_i$ to $N_j$
\End
\EndFor
\end{algorithmic}
\end{algorithm}

Step 1 can be carried out in $O(n + m + C)$ time by using radix sort with $n$ buckets
since for all $i, |C_i| \leq n$. In Step 2, we look at each $C_i$ once and we look at each species in
$C_i$ at most once. Thus, Step 2 takes $O(C)$ time. The remaining bookkeeping can also be
done in $O(C)$ time. Therefore, the running time of Phylogeny is $O(C)$, since usually
$n \ll m$ and $m \leq C$.

**Lemma 2** If Phylogeny returns Failure, then $C$ has no character tree.

**Proof:** Suppose Failure occurs when $C_i$ is being considered and we find that $A[y] \neq
A[x]$. Let $A[x] = N_j$. Then $C_i \cap C_j \neq \emptyset$ since $x \in C_i \cap C_j$. Since $C_j$ was considered before
$C_i$, $|C_j| \geq |C_i|$. By Lemma 1, $C$ has no character tree unless $C_j \supseteq C_i$.

Suppose $C_j \supseteq C_i$. We shall show that in this case there exists a $C_k$ such that $C_i$
and $C_k$ violate Lemma 1. Since $A[y] \neq N_j$ and $y \in C_j$, we must have $A[y] = N_k$ where $j < k < i$. Now, $C_i \cap C_k \neq \emptyset$ because $y \in C_i \cap C_j$. Since $A[x] = N_j$ and $k > j$, $x \not\in C_k$.
Therefore, $x \in C_i$, $x \not\in C_k$ and $C_k \not\supseteq C_i$. Since $|C_i| \leq |C_k|$, $C_i \not\supset C_k$ (we use $\subset$ and $\supset$ to
denote proper inclusion). Therefore, $C_i$ and $C_k$ violate Lemma 1 and $C$ has no character
tree. $\square$

**Lemma 3** If Phylogeny succeeds, then the tree constructed is a character tree for $C$.

**Proof:** Since we assume that all characters are nonempty, at the successful termination
of Phylogeny, all the nodes will be in the constructed tree, say $T$. Hence $T$ satisfies
(C1). We show that conditions (C2) and (C3) are maintained after each iteration of the
loop. Initially, these conditions are trivially true. Suppose they hold up to the $(r - 1)^{th}$
iteration of the loop where $r > 0$. In the $r^{th}$ iteration, we consider $C_r$. We argue that
conditions (C2) and (C3) hold after the completion of this iteration.

Suppose $N_p$ becomes the parent of $N_r$ and $C_r \not\supseteq C_p$. Then there exists a $y \in C_r$ such
that $y \not\subset C_p$. Thus, $A[y] \neq N_p$. Since $N_p$ is the parent of $N_r$, the first species $x$ picked
from $C_r$ has $A[x] = N_p$. Therefore, $A[y] \neq A[x]$ and \texttt{Phylogeny} would return \texttt{Failure}. Hence $C_r \subseteq C_p$ and (C2) is satisfied.

If $N_s$ has no siblings, then (C3) trivially holds true. Suppose $N_s$ has siblings and their parent is $N_p$. Let $x$ be the first species picked from $C_r$. Since $N_p$ is the parent of $N_s$, we have $A[x] = N_p$. Let $N_s$ be a sibling of $N_r$ such that $C_r \cap C_s \neq \emptyset$ and let $y \in C_r \cap C_s$. Then $A[y]$ is either $N_s$ or one of the descendants of $N_s$, but not $N_p$. Therefore, $A[y] \neq A[x]$ and \texttt{Phylogeny} would return \texttt{Failure}. Hence $C_r \cap C_s = \emptyset$ and (C3) is satisfied. □

3 An online algorithm for characters

It may sometimes be necessary to update an existing phylogeny or test if a newly introduced character is compatible with the current set of characters. In this section we consider an online algorithm for characters which answers the update and compatibility questions. Note that given a compatible set of characters $C$, a CT $R$ for $C$, and a character $C_t$, the character set $C' = C - \{C_t\}$ is also compatible and a CT $R'$ for $C'$ can easily be constructed from $R$ in $O(|C_t|)$ time. For this reason, in the remainder of this section, we concentrate on the problem of adding new characters.

In the online version for characters, characters arrive one at a time. Without loss of generality, we assume that characters are indexed $C_1, C_2, \ldots$, according to their order of arrival. As before, we stipulate the existence of a character $C_0$ which contains all $n$ species and that all characters are nonempty. The online binary phylogeny problem is as follows: Given a compatible set of characters $C_{t-1} = \{C_0, C_1, \ldots, C_{t-1}\}$ a character tree $R_{t-1}$ for $C_{t-1}$, and a character $C_t$, determine whether or not $C_t = C_{t-1} \cup \{C_t\}$ is a compatible set of characters and, if so, construct a character tree $R_t$ for $C_t$.

For convenience, we will work with a compact version of the character tree, which we shall refer to as a compact character tree (CCT). It is equivalent to a character tree but renders itself more easily to online modifications by identifying nodes with identical characters as a single node. Formally, a CCT for a compatible set of characters $C = \{C_0, \ldots, C_m\}$ is a tree $R$ with vertex set $V(R) = \{N_0, N_{j_1}, \ldots, N_{j_l}\}$, where $\{j_1, \ldots, j_l\} \subseteq \{1, \ldots, m\}$, together with a labeling function $label$ that maps every node in the tree to a subset of $\{1, \ldots, m\}$. $R$ and $label$ must satisfy the following properties:

1. $N_0$ is the root for $R$. ($C_0$ has all $n$ species.)
2. For every $N_i \in V(R)$, if $N_i$ has a parent $N_p$, then $C_i \subseteq C_p$.
3. For every $N_i \in V(R)$, $C_i \cap C_s = \emptyset$, for each sibling $N_s$ of $N_i$.
4. For every $N_j \in V(R)$, $label(N_j) = \{i : C_i = C_j\}$.
5. $\{label(N_j) : N_j \in V(R)\} = \{0, 0, \ldots, m\}$.

Figure 3 shows a CCT for the example in Figure 2. We have the following incremental version of Lemma 1, whose proof is omitted.

\textbf{Lemma 4} Suppose $C = \{C_0, \ldots, C_{t-1}\}$ has a CCT $R$ and $C_t$ is the next character. Then $C \cup \{C_t\}$ has a CCT $R'$ iff either
Figure 3: Compact character tree $R$ for $T$ in Figure 2

(E1) $C_t = C_p$ for some $C_p \in \mathcal{C}$, or

(E2) There exists $C_p \in \mathcal{C}$ such that $C_t \subseteq C_p$ and for each child $N_r$ of $N_p$ in $R$, $C_t \cap C_r = \emptyset$ or $C_t \supseteq C_r$.

Note that all characters $C_p$ which satisfy (E2) will be equal and therefore, by (D4) they will be contained in the label set of a unique node in $R$. Furthermore, if $\mathcal{C} \cup \{C_t\}$ has a CCT $R'$, then it can be obtained from $R$ by carrying out the following transformation:

- If (E1) is satisfied, then add $t$ to $\text{label}(N_p)$.
- If (E2) is satisfied, then create a node $N_t$, assign $\text{label}(N_t) = \{t\}$, and make $N_p$ the parent of $N_t$. The children of $N_t$ are those children $N_r$ of $N_p$, if any, such that $C_t \supseteq C_r$.

This transformation is carried out by algorithm \textsc{Online-Phylogeny}(R, C_t) given below which assumes that $R$ is not empty (i.e., contains at least the root node $N_0$) and $C_t \neq \emptyset$. Because of the online requirement, the nodes of the tree will have to carry more information to facilitate fast processing of new characters. Specifically, each node $N_i \in V(R)$ stores

1. the set $C_i$, denoted by $\text{set}(N_i)$,
2. $|C_i|$, the cardinality of $C_i$,
3. $\text{parent}(N_i)$, a pointer to the parent of $N_i$,
4. $\text{label}(N_i)$, the label set of $N_i$,
5. list of pointers to its children, and
6. for every $x \in C_i$, a variable $H_i[x]$. $H_i[x]$ is a pointer to the child of $N_i$ whose character contains the species $x$. If none of these contain $x$, then $H_i[x]$ is nil.
Algorithm Online-Phylogeny($R,C_t$)

begin
$R' ← R$
Let $v ← A[l]$ where $l ∈ C_t$ and $∀i ∈ C_t$, $|set(A[l])| ≥ |set(A[i])|$
while $|C_t| > |set(v)|$ do
$v ← parent(v)$
Let $v = N_p$  /* $|C_t| ≤ |C_p| */
(*)
if $C_t ∉ C_p$ then return FAILURE
else if $|C_t| = |C_p|$ then Add $t$ to $label(N_p)$; return $R'$
else begin
create a new child $N_t$ for $N_p$
$label(N_t) ← \{t\}; parent(N_t) ← N_p$
for every $x ∈ C_t$ do begin
$H_t[x] ← H_p[x]; H_p[x] ← N_t$
if $H_t[x] = \text{nil}$ then $A[x] ← N_t$
else begin
Let $H_t[x] = N_r$
if $parent(N_r) = N_p$ then if $C_r ⊂ C_t$
then begin
Add a link from $N_t$ to $N_r$
Remove the link from $N_p$ to $N_r$
$parent(N_r) ← N_t$
end
end
else return FAILURE
end
endfor
(S2)
return $R'$
end

Lemma 5 If Online-Phylogeny($R,C_t$) returns FAILURE, then $C \cup \{C_t\}$ has no CCT.

Proof: Suppose FAILURE occurs at (F1) when $|C_t| ≤ |C_p|$ but $C_t ∉ C_p$. Then neither one of $C_t, C_p$ contains the other. Since $N_p$ is an ancestor of $A[l], l ∈ C_p ∩ C_t$. Therefore, $C_t ∩ C_p ≠ \emptyset$ and $C_t, C_p$ violate Lemma 1; thus, $C \cup \{C_t\}$ has no CCT.

Suppose FAILURE occurs at (F2). Then $C_t ⊂ C_p$, but there exists $x ∈ C_t$ such that $H_t[x] = N_r ≠ \text{nil}, N_r$ is a child of $N_p$, and $C_r ∉ C_t$. We first show that the node $N_p$ found after the while loop is the only candidate for becoming the parent of $N_t$. Since $l$ must be in the character set of the parent of $N_t$ and $A[l]$ is the lowest node in $R$ containing $l$, the parent of $N_t$ lies on the path from $A[l]$ to the root of $R$. No node on the path from $A[l]$ to $N_p$ can be the parent of $N_t$ since the cardinality of its character set is smaller than $|C_t|$.
The nodes on the path from \( N_p \) to the root of \( R \) are not candidates for being a parent of \( N_i \) since they have a child which is neither disjoint from nor a subset of \( C_t \). Therefore, \( N_p \) is the only candidate for becoming parent of \( N_i \). Now, since \( N_r = H_t[x] \neq \text{nil}, x \in C_r \cap C_t \). Hence, if \( C_r \not\subseteq C_t \), (E2) is violated and \( C \cup \{ C_t \} \) has no CCT. □

**Lemma 6** If \( \text{Online-Phylogeny}(R,C_t) \) succeeds, then the tree \( R' \) constructed is a CCT for \( C \cup \{ C_t \} \).

**Proof:** If \( R' \) is returned at (S1), then there exists a \( C_p \in C \) such that \( C_p = C_t \). In this case, the algorithm modifies \( R \) by adding label \( t \) to \( \text{label}(N_p) \) in accordance with (D4). Since \( R \) satisfies (D1)-(D5), \( R' \) will also satisfy these conditions.

If \( R' \) is returned at (S2), then there is no \( C_p \in C \) such that \( C_p = C_t \). In this case, the algorithm adds a new node \( N_i \) to \( R \). \( N_i \) is made a child of a node \( N_p \) such that \( C_t \subseteq C_p \) and made the parent of zero or more nodes \( N_r \) such that \( N_r \) is a child of \( N_p \) in \( R \) and \( C_r \subseteq C_t \). Therefore, by (E2) \( R' \) is a CCT if \( C_t \) is disjoint from all \( C_r \) such that \( N_r \) is a child of \( N_p \) and \( C_r \not\subseteq C_t \). Now if \( x \in C_r \cap C_t \), then \( x \in \text{set}(H_t[x]) \cap C_r \). Since \( H_t[x] \) was a sibling of \( N_r \) in \( R \), we have a contradiction as \( R \) satisfies (D3). □

**Running time of Online-Phylogeny(\( R, C_t \))**: The time taken for finding the initial node \( v \) is \( O(|C_t|) \) since each node \( N_i \) stores \( |C_i| \). The number of iterations of the while loop is at most \( |C_t| \) since \( R \) is a CCT and by (D2), \( |\text{set}(\text{parent}(v))| \geq |\text{set}(v)| + 1 \). The only costly operation in the algorithm is checking for set containment in (\(*\)) and (\(\ast\ast\)). Clearly, (\(*\)) is carried out just once. For (\(\ast\ast\)), we have the following. Let \( N_{j_1}, \ldots, N_{j_k} \) be the children of \( N_i \). Note that \( \sum_{i=1}^k |C_{j_i}| \leq |C_t| \). For each child \( N_i \) of \( N_r \), the operation \( C_r \subseteq C_t \) is carried out precisely once when \( \text{parent}(N_r) \) is set to \( N_i \) and therefore, for any other \( x \in N_r \cap N_i \), the test for \( \text{parent}(N_r) = N_p \) fails. We shall now present two implementations for carrying out the subset checking and analyze the running time for each case.

**Sets as binary search trees**: Each node \( N_i \) stores the species in \( C_t \) as a binary search tree \( S_i \). The time required for checking whether \( C_t \subseteq C_p \) is at most \( O(|C_t| \log |C_p|) \) since for each element \( x \in C_t \), we need to check whether \( x \) is in \( S_p \) which requires \( O(\log |C_p|) \) time. Therefore, algorithm Online-Phylogeny(\( R, C_t \)) takes \( O(|C_t| \log |C_p|) \) time and by invoking it repeatedly, the phylogenetic tree can be found in \( O(C \log n) \) time.

**Sets as arrays**: Each node \( N_i \) contains an array \( S_i \) of length \( n \) where \( S_i[x] = 1 \) if \( x \in C_t \) and \( S_i[x] = 0 \) otherwise. \( S_i \) can be initialized in \( O(n) \) time. The node also stores species in \( C_t \) as a linked list \( L_i \). The time required to check whether \( C_t \subseteq C_p \) is \( O(|C_t|) \), since we need to examine each \( x \in L_i \) exactly once, to check whether \( S_p[x] = 1 \). Therefore, algorithm Online-Phylogeny(\( R, C_t \)) has a running time of \( O(n) \) and the array implementation will have a total running time of \( O(nm) \) for finding the phylogenetic tree.

4 An online algorithm for species

We now consider what, in a sense, is a dual version of the problem we studied in Section 3, namely, to dynamically maintain a CCT under a sequence of additions and deletions of
species. A species $S$ is represented by the set $I(S)$ of the indices of the characters exhibited by $S$. Given a set $S$ of species described in this way, there is a corresponding set $C$ of characters, where each $C_i \in C$ is given by $C_i = \{ S \in S : i \in I(S) \}$. We shall say that $S$ is compatible if the associated set of characters $C$ is compatible. As before, we shall assume that every species exhibits character $C_0$; i.e., $0 \in I(S)$ for all $S \in S$.

Suppose we have a compatible set of species $S = \{ S_1, S_2, \ldots, S_{n-1} \}$ and a CCT $R$ for $S$. Clearly, for any $S_i \in S$, the set of species $S' = S - \{ S_i \}$ is compatible. Constructing a CCT $R'$ for $S'$ simply requires removing $S_i$ from each node $N_j \in R$ such that $S_j \in C_j$.

All of these nodes must lie on a single path in $R$. The removal of $S_i$ may render identical two nodes which were formerly distinct sets $C_j$ and $C_k$ which differed only in $S_i$. Also, some set $C_j$ may become empty and thus, $N_j$ may have to be deleted from $R$. The key to updating $R$ efficiently therefore, lies in representing the nodes and their associated sets properly. For example, if sets are represented as boolean arrays of length $n$, removal of a species $S_i$ takes $O(1)$ time per set $C_j$ such that $j \in I(S_i)$, for a total of $O(|I(S_i)|)$ time. Using a binary search tree representation of sets, removal of $S_i$ from $C_j$ when $j \in I(S_i)$ takes $O(\log n)$ time, for a total of $O(|I(S_i)| \cdot \log n)$ time per deletion. All other adjustments to $R'$ can be done in $O(|I(S_i)|)$ time. For brevity, we shall leave the details to the reader and turn our attention to the problem of adding a species to $S$.

Suppose we wish to determine whether $S' = S \cup \{ S_i \}$ is compatible. Let $I(S_i) = \{ 0, i_1, i_2, \ldots, i_k \}$ be the indices of the characters of the new species $S_i$ and $R$ be the given tree; i.e., $S_i$ should be added to each of the characters in $\{ C_0, C_{i_1}, C_{i_2}, \ldots, C_{i_k} \}$. Our algorithm is based on the following result.

**Lemma 7** Suppose $S = \{ S_1, S_2, \ldots, S_{n-1} \}$ has a CCT $R$ and $S_i$ is the next species with $I(S_i) = \{ 0, i_1, \ldots, i_k \}$. Let $I_{old} = I(S_i) \cap I(S)$ where $I(S) = \cup S \in S I(S_i)$. Then $S \cup \{ S_i \}$ is a compatible set of species if there exists a path $P$ in $R$ from the root to some node $N_j \in V(R)$ such that both of the following conditions hold.

1. \( P_1 \) $I_{old} \subseteq \cup \{ \text{label}(N_i) : N_i \in P \}$.
2. \( P_2 \) label$(N_j) \cap I(S_i) \neq \emptyset$ and for each $N_i \in P - N_j$, label$(N_i) \subseteq I_{old}$.

**Proof:** We shall first argue that if there is no path $P$ satisfying (P1) and (P2), then $S \cup \{ S_i \}$ is incompatible. Let $P'$ be a longest path satisfying (P2). Note that there always exists such a path $P'$ since $0 \in I(S_i) \cap \text{label}(N_0)$ where $N_0$ is the root of $R$. Since $P'$ does not satisfy (P1) and every $i \in I(S)$ is present in the label set of exactly one node in $R$, there exist $N_j \in V(R)$ such that $N_j \notin P'$ and label$(N_j) \cap I(S_i) \neq \emptyset$. Suppose $N_j$ is the highest such node. We now have following cases to consider:

**Case 1:** Suppose parent$(N_j) \in P'$ and there exists a sibling $N_k$ of $N_j$ such that $N_k \in P'$. By definition, $C_j \cap C_k = \emptyset$. Since $\text{label}(N_j) \cap I(S_i) \neq \emptyset$ and $\text{label}(N_k) \cap I(S_i) \neq \emptyset$, $S_i$ must be added to both $C_j$ and $C_k$. Therefore, $C_j \cup \{ S_i \}$ and $C_k \cup \{ S_i \}$ violate Lemma 1.

**Case 2:** Suppose parent$(N_j) = N_p \in P'$ and $N_j$ has no sibling that lies on $P'$. Since $P'$ is of maximum length and $N_j \notin P'$, label$(N_p) \not\subseteq I_{old}$ and label$(N_p) \cap I_{old} \neq \emptyset$. By definition, $C_j \subseteq C_p$. Let $f \in \text{label}(N_p) - I_{old}$. Since $f \notin I(S_j)$ and $j \in I(S_i)$, $S_i$ gets added to $C_j$ but not to $C_f$. Therefore, $C_j \cup \{ S_i \}$ and $C_f$ violate Lemma 1.
Case 3: Suppose \( \text{parent}(N_j) = N_p \not\in P' \). Then \( \text{label}(N_p) \cap I(S_t) = \emptyset \) and \( S_t \) must be added to \( C_j \) but not to \( C_p \). But then, \( C_j \cup \{S_t\} \) and \( C_p \) violate Lemma 1.

In all cases, \( S \cup \{S_t\} \) is incompatible.

Now, suppose there exists a path \( P \) satisfying (P1) and (P2). We shall show that, in this case, we can build a CCT \( R' \) for \( S' = S \cup \{S_t\} \) and hence, that \( S' \) is compatible. As before, let \( N_l \) be the lowest node in \( P \).

First, let \( J_1 = \text{label}(N_l) \cap I(S_t) \) and \( J_2 = \text{label}(N_l) - J_1 \). If \( J_2 = \emptyset \), then add \( S_t \) to every \( N_i \in P \). Otherwise, split \( N_l \) into two nodes as shown in Figure 4. The top one will be \( N_{l_1} \), where \( l_1 \) is any element of \( J_1 \) and it will have \( \text{label}(N_{l_1}) = J_1 \). The other node will be \( N_{l_2} \), where \( l_2 \) is any element of \( J_2 \) and it will have \( \text{label}(N_{l_2}) = J_2 \). Next, if \( I_{old} \not= I(S_t) \), then choose any \( j \in I(S_i) - I_{old} \) and create a new node \( N_j \), which will be a child of \( N_{l_1} \). Define \( \text{label}(N_j) = I(S_t) - I_{old} \) and set \( (N_j) = \{S_t\} \). Finally, \( S_t \) is added to \( N_{l_1} \) and to every \( N_i \in P - N_l \). It is easy to verify that the resulting tree satisfies (D1)-(D5).

An algorithm for adding a species to a compatible set of species immediately follows from the proof of the previous lemma. Since there can be only one path \( P \) satisfying the conditions of the above lemma, we can find it as follows: First, we find a node \( N_i \in V(R) \) such that \( \text{label}(N_i) \cap I(S_t) \not= \emptyset \) and \( |C_i| \) is as small as possible. \( N_i \) will be the candidate for the lowest node in the path \( P \). Next, we go up the tree from \( N_i \), testing for each node \( N_i \) that we encounter whether \( \text{label}(N_i) \subseteq I_{old} \). If this does not hold, then we return FAILURE. Otherwise, we test whether the union of the labels we encountered contains \( I_{old} \) (note that this can actually be done as we are going up the tree). If not, then return
FAILURE. Otherwise, the path $P$ we have found satisfies the conditions of Lemma 7 and we modify the tree as specified in the proof of that lemma.

The lowest node $N_i$ as required above can be efficiently found by keeping an array $B[1..m]$ where $B[i]$ is a pointer to the node $N_j \in V(R)$ such that $i \in label(N_j)$. Most of the work done by the algorithm we have sketched involves testing for a given $N_i \in V(R)$ whether $label(N_i) \subseteq I_{\text{old}}$. The only significant problem is the representation of labels so that this test can be done efficiently. As done for set$(N_i)$ in Section 3, we can implement $label(N_i)$ as either a boolean array of length $m$ or as a binary search tree. The array implementation leads to a $O(m)$ running time per update and to a $O(nm)$ algorithm for adding $n$ species. The binary search tree implementation leads to a $O(|I(S_i)| \cdot \log m)$ running time per update and to a $O(C \log m)$ running time algorithm for adding $n$ species.

References


