Fast Computation of Supertrees for Compatible Phylogenies with Nested Taxa

Vincent Berry[1] and Charles Semple[2,3]

14 November 2005

[1] *Département Informatique, L.I.R.M.M. - C.N.R.S., 161 rue Ada, 34392 Montpellier Cedex 5, France*
vberry@lirmm.fr

[2] *Biomathematics Research Centre, Department of Mathematics and Statistics, University of Canterbury, Christchurch, New Zealand*
c.semple@math.canterbury.ac.nz

[3] Corresponding author.

Keywords. Phylogenetics, supertree methods, nested taxa, compatibility, strepsirrhine phylogeny.

Abstract

Typically, supertree methods combine a collection of source trees in which just the leaves are labelled by taxa. In such methods the resulting supertree is also leaf-labelled. An underlying assumption in these methods is that, across all trees in the collection, no two of the taxa are nested; for example, "buttercups" and "plants" are nested taxa. Motivated by Page, the first supertree algorithm for allowing the source trees to collectively have nested taxa is called AncestralBuild. Here, in addition to taxa labelling the leaves, the source trees may have taxa labelling some of their interior nodes. Taxa labelling interior nodes are at a higher taxonomic level than that of their descendants (for example, genera versus species). Analogous to the supertree method Build for deciding the compatibility of a collection of source trees in which just the leaves are labelled, AncestralBuild is a polynomial-time algorithm for deciding the compatibility of a collection of source trees in which some of the interior nodes are also labelled by taxa. Although a more general method, in this paper we show that the original description of AncestralBuild can be modified so that the running time is as fast as the current fastest running time for

Build. Fast computation for deciding compatibility is essential if one is to make use of phylogenetic databases that contain thousands of trees on tens of thousands of taxa. This is particularly so as AncestralBuild is incorporated as a basic tool inside more general supertree methods (that is, methods that always output a tree regardless of the compatibility of the source trees). We apply the method to propose a comprehensive phylogeny of the strepsirrhines, a major group of the primates.

Introduction

Supertree methods are a fundamental and practical way of inferring phylogenies. Generally speaking, these methods amalgamate a collection of "source" trees on overlapping subsets of taxa into a single parent tree that contains the taxa of all of the source trees. This parent tree is called a *supertree*. This approach to constructing evolutionary trees is particularly appealing because it allows the inference of an evolutionary scenario from a combination of analyses differing in the set of taxa they encompass as well as in the primary data from which they were conducted (for example, molecular or morphological studies). The increasing popularity of these methods and the diversity of ways in which they can be used is highlighted in a recently published book (Bininda-Emonds, 2004).

Historically, supertree methods only make use of the leaf labels of the source trees, with the resulting supertree having the property that taxa are only attached at leaves. On this basis, supertree methods can deal with taxa at different taxonomic levels only in the case where the leaves of the source trees represent *non-nested* taxa (*e.g.*, "elephant" and "mammal" should not be represented as two distinct leaves amongst the source trees as the former taxa is nested inside the latter taxa). However, in supertree studies this situation does occur, especially concerning outgroups. For example, a source tree containing strepsirrhine species could include Haplorrhini, a sub-order, as an outgroup leaf, and another source tree on primates would include several haplorrhine species as leaves, while the latter are nested inside the former taxon. Because of the assumption that taxa of the source trees are non-nested, traditional supertree methods produce in that case an incorrect supertree. Thus, unless one resorts to preprocessing of source trees to remove nested taxa via taxonomic synonymy (Bininda-Emonds et al., 2004), traditional supertree methods have limited use in the case of nested taxa, as arises when one is to amalgamate trees from phylogenetic databases such as TreeBASE (Sanderson et al., 1993). This was recently highlighted by Page (2004). Indeed, TreeBASE incorporates trees from many different published phylogenetic studies that focus on a variety of different biological problems, and hence consider evolutionary relationships at different taxonomic levels.

As a consequence of this limitation, Page (2004) motivated the task of designing supertree methods that allow the interior nodes as well as all of the leaves to represent taxa in the source trees. In these *nested-taxa* source trees, an interior label corresponds to a taxon at a higher taxonomic level than any of its descendants. Note that interior labels can be either available initially in the source trees, or added to some source trees based on taxa present in other source trees at the time the collection of studied source trees is formed. The first computational problem that arises is to find a polynomial-time algorithm for deciding whether or not a collection of nested-taxa source trees is compatible and, if so, constructing an appropriate supertree.

In answer to this problem, Daniel and Semple (2004) provided an algorithm called AncestralBuild. This algorithm generalizes Build, one of the first supertree methods (Aho et al., 1981). The polynomial-time algorithm Build takes a collection $\mathcal{P}$ of rooted

leaf-labelled trees as input and decides whether or not $\mathcal{P}$ is compatible, in which case it returns a leaf-labelled supertree that *displays* $\mathcal{P}$. That is, the supertree preserves all of the relative groupings of taxa present in the source trees. For a collection of nested-taxa trees, if a supertree preserves all ancestral relationships as well as all groupings of taxa, then the supertree is said to *ancestrally display* this collection and the collection is said to be *ancestrally compatible*. These concepts are formally defined in the next section. However, we make a comment here on the way in which *ancestor* is used in this paper: by writing that a taxon t is an ancestor of a taxon t', we mean that t is either a hypothetical ancestral taxon of t' or that t is the name of a taxonomic grouping containing t', so that node labelled t is ancestral to the node labelled t'. The algorithm AncestralBuild takes a collection $\mathcal{P}$ of nested-taxa trees as input and outputs a supertree that ancestrally displays $\mathcal{P}$ if such a supertree exists, otherwise it states that the collection is not ancestrally compatible. Though designed to handle trees containing taxa at both internal nodes and leaves, AncestralBuild also accepts collections of source trees that have taxa only at the leaves, because leaf-labelled trees are a special case of nested-taxa trees. In that particular case, AncestralBuild decides the compatibility of the source trees in the usual sense. Consequently, it does indeed generalize Build.

AncestralBuild has the desirable property to give an exact answer in polynomial-time (Daniel and Semple, 2004). However, there can be three objections to its use. First, for incompatible phylogenies to be combined, an all-or-nothing algorithm, that is just stating the incompatibility when it arises, is not desirable. Second, even for the easiest case of source trees that are all fully-resolved and have taxa only at the leaves, the running time of the version of AncestralBuild stated in Daniel and Semple (2004) is $O(t^2n^3)$, where t is the number of source trees and n is the number of taxa. Despite being polynomial, this running time makes AncestralBuild a reasonably slow algorithm in practice, particularly if it is to handle the thousands of trees stored in databases such as TreeBASE. Third, in the absence of internal labels in the source trees, the method will build a nonsensical supertree, not knowing for instance that the "mammal" taxon met in one tree is an ancestor of "elephant", present in another tree. We answer these three objections below.

Inferring a supertree for incompatible source trees. Primary data sequences are getting longer and phylogenetic methods more accurate, so that a reasonable number of published phylogenies are compatible. For example, Llabrés et al. (2005) found a very low rate of incompatible trees in TreeBASE. However, it is still relatively frequent that one has to deal with incompatible source trees. For example, it is well-known that gene trees may conflict with species trees (due to *e.g.*, lateral gene transfer or hybridization), and that some trees may contain erroneous branches inferred due to noisy information for some internal edges in the primary data. In the former case, one may want to construct a supertree to identify the disagreements between the gene trees and the species trees. In such cases, an all-or-nothing algorithm may not be enough and one can understandably prefer a more general supertree method that outputs a supertree that either conflicts with or omits some information present in the source trees. However, for any general supertree method, a

basic property that one would always like is that of *consistency*; that is, if the source trees carry no conflicting information, then the supertree returned by the method displays each of the source trees. Because the property of consistency is such a compelling property, many general supertree methods dealing with leaf-labelled trees (respectively, nested-taxa trees) are likely either to have BUILD (respectively, ANCESTRALBUILD) as a subroutine or to be a variant of BUILD (respectively, ANCESTRALBUILD). Indeed, this is already the case for some general methods: both MINCUTSUPERTREE (Semple and Steel, 2000) method and its modified version (Page, 2002) are variants of BUILD and, more recently, Daniel and Semple (2005) describe a class of general supertree methods for nested-taxa source trees that is a variant of ANCESTRALBUILD. (This class and more particularly the underlying general supertree method NESTEDSUPERTREE is described further in the last section.) Moreover, these all-or-nothing algorithms can be repeatedly used in simple schemes to extract compatible parts out of a collection of incompatible source trees. We highlight two examples of such schemes in the discussion part of this paper.

Reasonable running time. Given the amount of information in current tree databases, it is not unreasonable to try to amalgamate hundreds of trees that collectively contain thousands of taxa and, consequently, fast algorithms are essential. To deal with fully-resolved (*i.e.*, *binary*) leaf-labelled trees, Henzinger et al. (1999) proposed a fast implementation of BUILD that runs in $O(mn^{\frac{1}{2}})$ time, where m is the total sum of the number of nodes in each of the source trees and n is the total number of taxa. (They also proposed a variant of BUILD running in $O(m + n^2 \log n)$ time but, in the case of compatibility, it typically resulted in a supertree with edges not supported by any part of the data, an undesirable feature for a supertree algorithm.) For comparison between this running time and the $O(t^2n^3)$ running time of ANCESTRALBUILD, note that m can range from $O(n)$ to $O(tn)$. Thus, the running time of ANCESTRALBUILD as stated in Daniel and Semple (2004) is still relatively slow compared to the implementation provided by Henzinger et al. (1999) for BUILD in the special case of fully-resolved leaf-labelled trees. However, comparing the gap between these running times provides hope for improving ANCESTRALBUILD on this aspect, despite the latter being a much more general algorithm for it allows nested-taxa trees and trees that are partially resolved. In this paper we describe several modifications to ANCESTRALBUILD that greatly reduce its computational complexity. Some of these modifications have a similar flavour to techniques used in Henzinger et al. (1999). The running time of ANCESTRALBUILD resulting from these modifications is as fast as the current fastest algorithm for source trees in which only the leaves are labelled (Henzinger et al., 1999). More particularly, the achieved running time is almost linear in the size of the source trees, when these trees are fully-resolved. Furthermore, extending the implementation of Henzinger et al. (1999) for BUILD in the most canonical and efficient way to allow partially-resolved leaf-labelled trees in its input, we also show that the running time of our modification of ANCESTRALBUILD is the same as that for this extension of BUILD.

Having a sensible supertree as output. In the case where nested taxa are present in different source trees but these trees contain no internal label, then, unless extra information is provided to recognize such nestings, there is no chance that AncestralBuild (or any supertree method) produces a sensible supertree. For example, if one tree contains "elephant" and another "mammal" without any tree indicating that the latter is an ancestor of the former, then the supertree output by the methods will have both taxa as tips. This situation arises because of lack of information in the source trees. Unfortunatly, this will be a frequent situation in practice because already inferred trees are usually stored without internal labels, though this is allowed by the Newick format. In such a case, all that is required for the method to produce a sensible supertree is a (partial) reference taxonomy indicating the nestings between taxa of interest. This taxonomy can be included as one extra source tree or be divided into several extra very simple trees of two vertices, where an ancestor taxon is above one of its descendant taxon. This reference taxonomy can be either built by hand, or generated automatically by a program searching taxon names in web accessible taxonomies, like NCBI's *Taxonomy Browser* (`http://www.ncbi.nlm.nih.gov/entrez`) or the *Tree of Life* web site (`http://tolweb.org/tree`). See Page and Valiente (2005) and Hibbett et al. (2005) for programs filling aims close to that one.

To describe an application of AncestralBuild, we studied the evolutionary relationships of the Strepsirrhini, one of the two major primate groups. Monophyly of this group is widely accepted but some intrasubordinal relationships were still recently under debate. Recent studies investigating strepsirrhine phylogeny involve quite different kinds of data, including mtDNA sequences (Yoder et al., 2000), craniodental morphological data (Masters and Brothers, 2002) and retroposon analysis (Roos et al., 2004). We applied AncestralBuild to four phylogenies published in these papers, covering many different taxonomic levels from order to individuals. The method shows that these phylogenies are ancestrally compatible and provides as result a comprehensive supertree of the Strepsirrhini. This highlights, once again, the interest in the supertree approach, that allows to combine phylogenies inferred from different kinds of primary data.

The rest of the paper is organized as follows. The next section contains some preliminaries that are used throughout the paper. The following three sections contain a description of AncestralBuild as stated in Daniel and Semple (2004) and descriptions of our modifications of this algorithm as well as the running time of this modification. An application of AncestralBuild to obtain a strepsirrhine supertree is described in the following section. In addition to deciding compatibility, AncestralBuild can be used in a variety of ways for constructing supertrees from incompatible source trees. We highlight some of these ways in the last section. Finally, the appendix contains the proof which establishes that our modification of AncestralBuild does indeed decide ancestral compatibility for a collection of nested-taxa trees as well as a pseudo-code description of this algorithm.

We end the introduction by noting that a novel approach for testing the ancestral compatibility of two rooted semi-labelled trees has been recently proposed by Llabrés

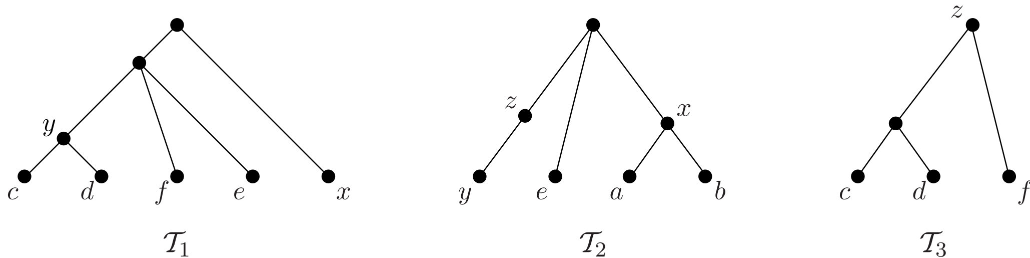

Figure 1: A compatible collection $\mathcal{P}$ of rooted semi-labelled trees.

et al. (2005).

Preliminaries

In this section, we describe some concepts that are frequently used in the paper. For further details, we refer the interested reader to Semple and Steel (2003).

Phylogenies. The *degree* of a node v in a graph (or, in particular, a tree) is the number of edges incident with v. We denote the degree of v by $d(v)$. Essentially, a rooted phylogenetic X-tree is a rooted tree whose leaves are labelled with the elements of a set X of taxa. We formally define it as follows. A *rooted phylogenetic tree (on X)* is an ordered pair $(T;\phi)$ consisting of a rooted tree T in which all interior nodes have degree at least three except the root which has degree at least two, and a map ϕ from X to the leaf set of T that assigns each element in X to a leaf in T so that no two elements are assigned the same leaf in T and each leaf of T is assigned an element of X. A rooted phylogenetic tree is fully-resolved or *binary* if the root of T has degree two and all other interior nodes of T have degree three.

The concept of rooted phylogenetic trees naturally extends to trees in which some of the interior nodes are also labelled by taxa. Such trees, called nested-taxa trees or rooted semi-labelled trees, are used to display in the same tree taxa at different taxonomic levels. More precisely, a *rooted semi-labelled tree* $\mathcal{T}$ on a taxa set X is an ordered pair $(T;\phi)$ consisting of a rooted tree T with root node ρ, and a map ϕ from X into the node set V of T such that, for all non-root nodes v of degree at most two, ϕ assigns v an element of X, and if ρ has degree zero or one, then ϕ also assigns ρ an element of X. If every node of T is assigned at most one taxa of X under ϕ, then $\mathcal{T}$ is said to be *singularly labelled*. Furthermore, if $\mathcal{T}$ is singularly labelled and the degree of any node in T is at most three except for the root which has degree at most two, then we say that $\mathcal{T}$ is *binary*. In Figure 1, each of $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$ is a rooted semi-labelled tree, but $\mathcal{T}_3$ is the only rooted semi-labelled tree that is binary.

Remarks.

1. We will often write a rooted semi-labelled X-tree for a rooted semi-labelled tree on X.

2. Observe that rooted phylogenetic trees are special types of rooted semi-labelled trees.

3. To simplify matters and because we see no practical reason for nodes in the source trees to be assigned more than one taxa of X, we will assume throughout the paper that all rooted semi-labelled trees that are source trees are singularly labelled. However, we note that the upgrade of the results in this paper to non-singular rooted semi-labelled trees is straightforward. Note that this remark about singular labelling does not apply to the output tree, where it is quite possible that some nodes are labelled with more than one taxa. For instance, a node joining human and chimp on a source tree that contains no other mammals could be equally labelled as anything from "hominoid" to "primate" to "mammal". This multiple listing of a node then becomes very important if all these names appear on other source trees and there is no user-input taxonomy to sort this out.

Let $\mathcal{T} = (T; \phi)$ be a rooted semi-labelled tree on X. The set X is called the *label set* of $\mathcal{T}$ and we call the elements of X *labels*. We also use $\mathcal{L}(\mathcal{T})$ to denote the label set of $\mathcal{T}$. For example, in Figure 1, $\mathcal{L}(\mathcal{T}_1) = \{c, d, e, f, x, y\}$. For a node v of T, we denote the set of elements of X that are assigned to v by $\phi^{-1}(v)$ and say that the elements in $\phi^{-1}(v)$ *label* v. Furthermore, $\mathcal{T}$ is *fully labelled* if every node of T is labelled by an element of X. For a collection $\mathcal{P}$ of rooted semi-labelled trees, we denote the union of the label sets of the trees in $\mathcal{P}$ by $\mathcal{L}(\mathcal{P})$. Thus, for example, for the collection $\mathcal{P}$ of rooted semi-labelled trees shown in Figure 1, we have $\mathcal{L}(\mathcal{P}) = \{a, b, c, d, e, f, x, y, z\}$.

Let $\mathcal{T} = (T; \phi)$ be a rooted semi-labelled tree, and let $a, b \in \mathcal{L}(\mathcal{T})$. We say that a is a *descendant (label)* of b (or, alternatively, b is an *ancestor (label)* of a) if the path from the root of T to $\phi(a)$ includes $\phi(b)$. Furthermore, a and b are *not comparable* if neither a is a descendant of b nor b is a descendant of a. Now suppose that a and b are not comparable in $\mathcal{T}$. Then the node of T that is the last common node on the paths from the root of T to $\phi(a)$ and from the root of T to $\phi(b)$ is called the *most recent common ancestor* of a and b, and is denoted $\mathrm{mrca}_{\mathcal{T}}(a, b)$.

Lastly, throughout the paper, for a rooted semi-labelled tree $\mathcal{T}$, we will use $|\mathcal{T}|$ to denote the number of nodes in $\mathcal{T}$. Furthermore, for a collection $\mathcal{P}$ of rooted semi-labelled trees, we will use m to denote $\sum_{\mathcal{T} \in \mathcal{P}} |\mathcal{T}|$.

Compatibility. We will describe the notion of compatibility for rooted phylogenetic trees first, before extending this notion to rooted semi-labelled trees.

Let $\mathcal{T}$ be a rooted phylogenetic tree on X and let $\mathcal{T}'$ be a rooted phylogenetic tree on X', where X is a subset of X'. We say that $\mathcal{T}'$ *displays* $\mathcal{T}$ if, up to suppressing all non-root nodes of degree-two, the minimal rooted subtree of $\mathcal{T}'$ that connects the elements in X is a *refinement* of $\mathcal{T}$, that is $\mathcal{T}$ can be obtained from it by contracting edges. *Suppressing a degree-two node* v means replacing v and its two incident edges with a single edge. Observe

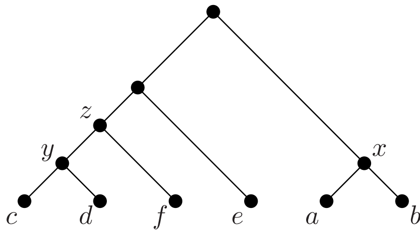

Figure 2: A rooted semi-labelled tree that ancestrally displays the collection $\mathcal{P}$ of trees shown in Figure 1. Indeed, as we shall soon see, this is the tree outputted by ANCESTRAL-BUILD when applied to $\mathcal{P}$.

that the use of "refinement" means that we allow for the resolution of soft polytomies in the definition of displays, where we recall that a soft polytomy represents an uncertainty as to the exact order of speciation as oppose to a certainty of a multiple speciation event. A collection $\mathcal{P}$ of rooted phylogenetic trees are *compatible* if there is a rooted phylogenetic tree $\mathcal{T}$ that simultaneously displays each of the trees in $\mathcal{P}$, in which case we say that $\mathcal{T}$ *displays* $\mathcal{P}$.

The notion of displays for rooted semi-labelled trees extends the notion of displays for rooted phylogenetic trees. In particular, let $\mathcal{T}$ be a rooted semi-labelled X-tree and let $\mathcal{T}'$ be a rooted semi-labelled X'-tree, where X is a subset of X'. Then $\mathcal{T}'$ *ancestrally displays* $\mathcal{T}$ if, up to suppressing non-root nodes of degree-two, the minimal rooted subtree of $\mathcal{T}'$ that connects the elements in X is a refinement of $\mathcal{T}$ and, for all $a, b \in X$, whenever a is a descendant label of b in $\mathcal{T}$, we have that a is a descendant label of b in this rooted subtree. A collection $\mathcal{P}$ of rooted semi-labelled trees is *ancestrally compatible* if there is a rooted semi-labelled tree $\mathcal{T}$ that ancestrally displays each of the trees in $\mathcal{P}$, in which case we say that $\mathcal{T}$ *ancestrally displays* $\mathcal{P}$. To illustrate this notion, the collection of trees in Figure 1 is ancestrally compatible, as each of its trees is ancestrally displayed by the rooted semi-labelled tree shown in Figure 2.

It is important to note that a collection of rooted semi-labelled trees may be ancestrally incompatible because of their interior labels and not their leaf labels. This could be as simple as a contradiction on an ancestor-descendant relationship, or as complex as the incompatibility of a set of most recent common ancestor relationships.

Graphs and Digraphs. Let G be a graph with node set V. We frequently use the notation $\{u, v\}$ to denote the edge joining the nodes u and v. Furthermore, the *connected components* of G are the subgraphs of G such that, for all $u, v \in V$, u and v are in the same connected component if and only if there is a path from u to v.

A *directed graph*, also called a digraph, is simply a graph in which, instead of having edges joining two nodes, we have *directed edges*; that is, edges directed from one node to another. Directed edges are also called *arcs*. We use the notation (u, v) to denote the arc directed from u to v. For a directed graph D, the *indegree* of a node v is the number of arcs

directed into v and the *outdegree* of v is the number of arcs directed out of v. Analogous to the connected components of an ordinary graph, the *(connected) arc components* of D are the sub-digraphs of D such that, for all $u, v \in V$, u and v are in the same arc component if and only if, ignoring the direction of the arcs, there is a path between u and v.

For the purposes of this paper, we say that a graph is *mixed* if it contains both arcs and edges. An *arc component* of a mixed graph is an arc component of the digraph obtained when masking the edges of this graph.

Let D be a (mixed) graph, let v be a node of D, and let U be a subset of the set of nodes of D. The (mixed) graph obtained from D by deleting v and each of its incident arcs and edges is denoted by $D\backslash v$. In general, we use $D\backslash U$ to denote the graph obtained from D by deleting each of the nodes in U. Furthermore, if V is the node set of D, then the *restriction of D to U* (also called the *subgraph of D induced by U*) is the subgraph of D that is obtained by deleting each of the nodes in $V - U$. This subgraph is denoted by $D|U$.

Finally, one typically views a rooted tree as an undirected graph. However, it will often be convenient in this paper to view a rooted tree as a directed graph where each edge is replaced with an arc directed away from the root.

AncestralBuild

For completeness and to make a comparison of the modifications we describe in this paper, in this section we give a full description of AncestralBuild as it is stated in Daniel and Semple (2004).

We begin by describing a construction on a collection of rooted semi-labelled trees, and a particular mixed graph on a collection of rooted fully-labelled trees. Both the construction and mixed graph are central to AncestralBuild.

Let $\mathcal{T} = (T; \phi)$ be a rooted semi-labelled tree on X, where T has node set V. We say that a rooted fully-labelled tree $\mathcal{T}'$ has been obtained from $\mathcal{T}$ by *adding distinct new labels* if, for each node of T that is not assigned a label under ϕ, we assign it an arbitrary label not in X so that no two new labels are the same. In general, if $\mathcal{P}$ is a collection of rooted semi-labelled trees, we say that $\mathcal{P}'$ has been obtained from $\mathcal{P}$ by *adding distinct new labels* if it has been obtained by adding distinct new labels to each tree in $\mathcal{P}$ so that across all trees in $\mathcal{P}'$ no two new labels are the same.

Example 1 To illustrate the above construction, let $\mathcal{P}$ be the collection of rooted semi-labelled trees shown in Figure 1. The collection $\mathcal{P}'$ shown in Figure 3 has been obtained from $\mathcal{P}$ by adding distinct new labels. Note that the added labels only need to be unique,

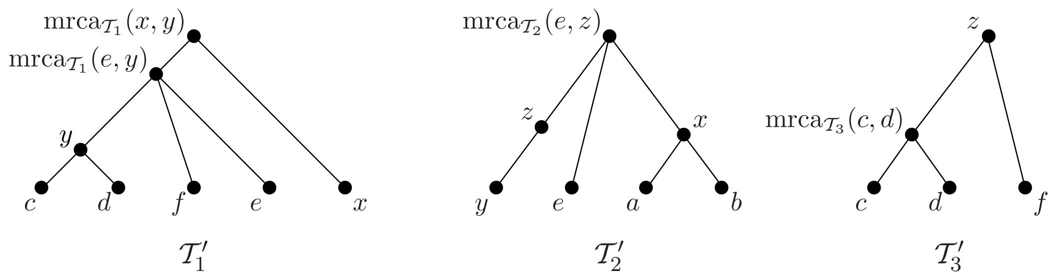

Figure 3: A collection $\mathcal{P}'$ of rooted fully-labelled trees.

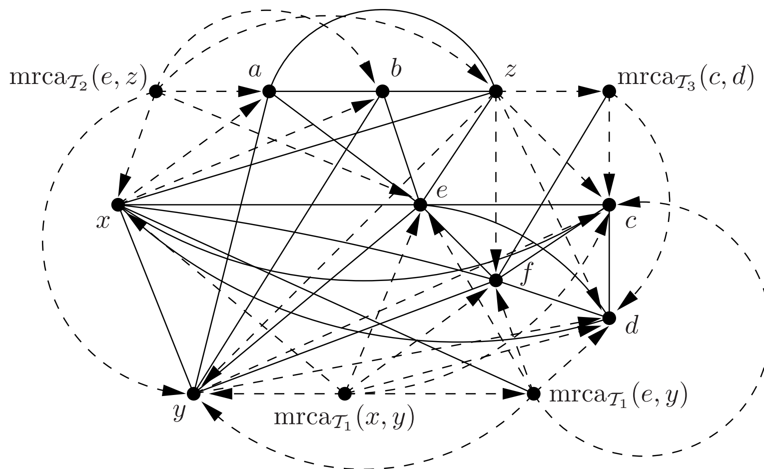

Figure 4: The descendancy graph of $\mathcal{P}'$. Arcs are shown as dashed lines with an arrow showing the direction of the arc, while edges are shown as solid lines.

and not as involved as the ones shown in Figure 3. The reason for choosing these particular labels will be made clear in the next section. □

Let $\mathcal{P}'$ be a collection of rooted fully-labelled trees. The *descendancy graph of* $\mathcal{P}'$, denoted $D(\mathcal{P}')$, is the mixed graph whose node set is $\mathcal{L}(\mathcal{P}')$, and whose arc and edge sets are

$$\big\{(c, a) : a \text{ is a descendant label of } c \text{ in some } \mathcal{T} \text{ in } \mathcal{P}'\big\}$$

and

$$\big\{\{a, b\} : a \text{ is not comparable to } b \text{ in some } \mathcal{T} \text{ in } \mathcal{P}'\big\},$$

respectively. Figure 4 shows the descendancy graph corresponding to the collection $\mathcal{P}'$ of rooted fully-labelled trees shown in Figure 3.

We now describe AncestralBuild. All of the work in the algorithm is performed by a subroutine called Descendant which decides the ancestral compatibility of a collection $\mathcal{P}'$ of rooted fully-labelled trees that has been obtained from the original collection $\mathcal{P}$ of rooted semi-labelled trees by adding distinct new labels. Loosely speaking, Descendant attempts to construct a rooted fully-labelled tree that ancestrally displays $\mathcal{P}'$ beginning

with the cluster $\mathcal{L}(\mathcal{P}')$ and successively breaking it down into disjoint subclusters. The way in which the clusters are broken up is decided by the descendancy graph which itself is successively broken into node induced subgraphs. The algorithm either completes the construction of such a tree or returns *not ancestrally compatible* if at some iteration the associated node induced subgraph of the descendancy graph has no nodes which have indegree zero and no incident edges.

Algorithm: ANCESTRALBUILD($\mathcal{P}$)
Input: A collection $\mathcal{P}$ of rooted semi-labelled trees on X.
Output: A rooted semi-labelled tree $\mathcal{T}$ that ancestrally displays $\mathcal{P}$ or the statement $\mathcal{P}$ *is not ancestrally compatible*.

1. Construct a collection $\mathcal{P}'$ of rooted fully-labelled trees from $\mathcal{P}$ by adding distinct new labels to the unlabelled nodes in the trees of the collection.

2. Construct the descendancy graph $D(\mathcal{P}')$ of $\mathcal{P}'$.

3. Call the subroutine DESCENDANT($D(\mathcal{P}')$).

4. If DESCENDANT returns *no possible labelling*, then return $\mathcal{P}$ *is not ancestrally compatible*. Otherwise, return the semi-labelled tree $\mathcal{T}'$ returned by DESCENDANT with the added labels removed.

Algorithm: DESCENDANT($D(\mathcal{P}')$)
Input: The descendancy graph of a collection $\mathcal{P}'$ of rooted fully-labelled trees.
Output: A rooted fully-labelled tree $\mathcal{T}'$ with root node v' that ancestrally displays $\mathcal{P}'$ or the statement *no possible labelling*.

1. Let $\mathcal{S}_0$ denote the set of nodes of $D(\mathcal{P}')$ that have indegree zero and no incident edges. If $\mathcal{S}_0$ is empty, then halt and return *no possible labelling*.

2. If $\mathcal{S}_0$ comprises of exactly one node labelled ℓ with outdegree zero, then return the tree composed of just one leaf labelled ℓ.

3. Otherwise,

- **(a)** Delete the elements of $\mathcal{S}_0$ (and their incident arcs) from $D(\mathcal{P}')$ and denote the resulting graph by $D(\mathcal{P}')\backslash\mathcal{S}_0$.
- **(b)** Find the node sets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k$ of the arc components of $D(\mathcal{P}')\backslash\mathcal{S}_0$.
- **(c)** Delete all edges of $D(\mathcal{P}')\backslash\mathcal{S}_0$ whose end nodes are in distinct arc components of this graph.

4. For each element $i \in \{1, 2, \ldots, k\}$, call DESCENDANT($D(\mathcal{P}')|\mathcal{S}_i$). If any of these calls return *no possible labelling*, then return this message. Otherwise, return the tree whose root node is labelled by $\mathcal{S}_0$ and which has $\mathcal{T}'_1, \ldots, \mathcal{T}'_k$ (the trees returned by the recursive calls) as child subtrees.

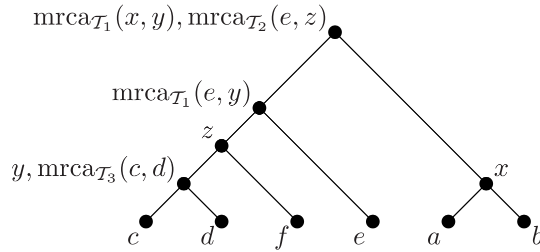

Figure 5: The rooted semi-labelled tree returned by DESCENDANT as described in Example 2.

Remark.

1. With respect to descendancy, the added labels act as necessary "place holders" for unlabelled nodes.

2. The recursive calls performed at Step 4 in DESCENDANT consider disjoint node induced subgraphs so that the processes applied to these subgraphs in subsequent iterations are independent from one subgraph to another.

Example 2 As an example of ANCESTRALBUILD applied to a collection of rooted semi-labelled trees, consider the collection $\mathcal{P}$ of trees shown in Figure 1. Suppose that Step 1 constructs the collection $\mathcal{P}'$ of rooted fully-labelled trees shown in Figure 3. Now Step 2 builds the descendancy graph $D(\mathcal{P}')$ as shown in Figure 4 . On the first iteration of DESCENDANT, Step 1 finds $\mathrm{mrca}_{\mathcal{T}_1}(x,y)$ and $\mathrm{mrca}_{\mathcal{T}_2}(e,z)$ as the only nodes of $D(\mathcal{P}')$ that have indegree zero and no incident edges. Deleting these elements in Step 3 results in the creation of two arc components, one containing nodes a, b, and x and the other containing the nine remaining nodes. Recursive calls to DESCENDANT investigate these two components separately. At Step 4 of the initial call to DESCENDANT, the subtrees returned by the two recursive calls are used as child subtrees of the root of the tree returned there. The root of this tree, which is shown in Figure 5, is labelled by $\mathcal{S}_0 = \{\mathrm{mrca}_{\mathcal{T}_1}(x,y), \mathrm{mrca}_{\mathcal{T}_2}(e,z)\}$. All added labels are eventually removed in Step 4 of ANCESTRALBUILD. The final tree returned by ANCESTRALBUILD applied to $\mathcal{P}$ is shown in Figure 2. □

The running time of ANCESTRALBUILD as it is stated above (and thus in Daniel and Semple (2004)) is given in Proposition 3.

Proposition 3 *Let $\mathcal{P}$ be a collection of rooted semi-labelled trees with $|\mathcal{P}| = t$ and $|\mathcal{L}(\mathcal{P})| = n$. Then* ANCESTRALBUILD($\mathcal{P}$) *runs in time $O(t^2n^3)$.*

Proof. Recall that $m = \sum_{\mathcal{T}\in\mathcal{P}} |\mathcal{T}|$. First note that the descendancy graph $D(\mathcal{P})$ contains $O(m)$ nodes and $O(tn^2)$ arcs and edges. In the worst case, every execution of DESCENDANT

removes only one node in $D(\mathcal{P})$, in which case the subroutine is executed $O(m)$ times. The computation time is dominated by the cost of Steps 3(b) and 3(c) in the subroutine. Finding the connected arc components of a digraph is linear in the number of its nodes and arcs. Thus, assuming that in the worst case only a constant number of edges are removed with each node, an execution of Step 3(b) can require up to $O(tn^2)$ time to process the restriction of $D(\mathcal{P})$ it is considering. Because of the m executions of the subroutine, this leads to an overall running time of $O(mtn^2)$ for Step 3(b). Finding edges across different arc components in Step 3(c) can necessitate at worst to examine the $O(tn^2)$ edges of the graph. This leads to an overall running time of $O(mtn^2)$ for Step 3(c). Noting that $m = O(tn)$ gives the final result. □

Remark. It is worth noting that the running time of AncestralBuild does not improve when each of the trees in $\mathcal{P}$ are fully-resolved and phylogenetic. Indeed, the descendancy graph still has $O(tn^2)$ arcs and edges in such cases.

Daniel and Semple (2004) were only interested in making sure that AncestralBuild runs in time polynomial in the size of the input. Indeed, other than providing a brief check to note that it is polynomial, no consideration to the actual running time is given. Consequently, some simple and not-so simple improvements to the algorithm were overlooked. In the next section we show how this running time can be reduced to almost linear running time. This improvement results from three changes in the algorithm: (i) drastically reducing the number of arcs and edges in the descendancy graph; (ii) using an ad-hoc graph of smaller size to compute the arc components of the various restrictions of the descendancy graph and identify edges between them; (iii) working these graph restrictions as actual subgraphs of the initial graph (and not as copies of bits of it), while maintaining node-connectivity through an efficient dynamic data structure. This data structure facilitates the discovery of new arc components resulting from edge deletions.

Improving the Running Time of AncestralBuild

In this section we describe our modification of AncestralBuild.

Reducing the size of the descendancy graph

We first show that a large amount of information included in the descendancy graph is redundant in the sense that the correctness of the algorithm is maintained when using a restricted version of this graph. For a collection $\mathcal{P}'$ of rooted fully-labelled trees, let $D^*(\mathcal{P}')$ be the graph having the same node set as $D(\mathcal{P}')$, but whose arc and edge sets are

$$\big\{(c, a) : \; a \text{ is a descendant label of } c \text{ in some } \mathcal{T} \text{ in } \mathcal{P}' \text{ such that there is no } b \in \mathcal{L}(\mathcal{T}) - \{a, c\} \text{ with } b \text{ a descendant of } c \text{ and } b \text{ an ancestor of } a\big\}$$

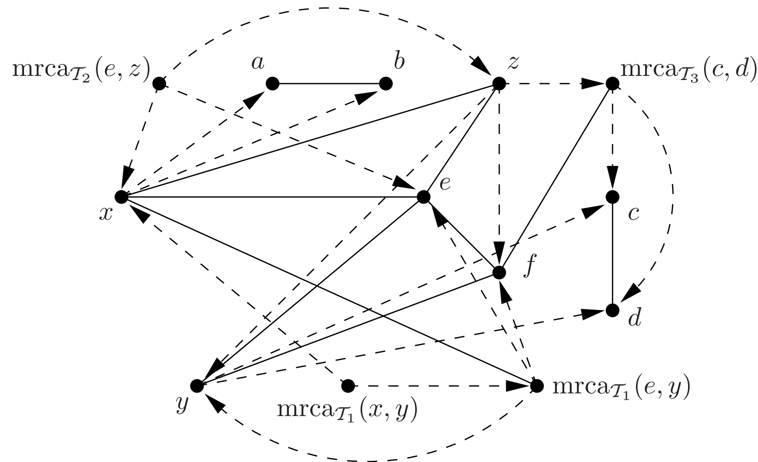

Figure 6: The restricted descendancy graph of $\mathcal{P}'$. Arcs are shown as dashed lines with an arrow showing the direction of the arc, while edges are shown as solid lines.

and

$$\big\{\{a,b\} : a \text{ is not comparable to } b \text{ in some } \mathcal{T} \text{ in } \mathcal{P}' \text{ such that there is a } c \in \mathcal{L}(\mathcal{T}) \text{ with } c \text{ the immediate ancestor label of both } a \text{ and } b\big\},$$

respectively. Clearly, the arc and edge sets of $D^*(\mathcal{P}')$ are subsets of the arc and edge sets of $D(\mathcal{P}')$, respectively, and so we call $D^*(\mathcal{P}')$ the *restricted descendancy graph of* $\mathcal{P}'$. The restricted descendancy graph of the collection $\mathcal{P}'$ shown in Figure 3 is shown in Figure 6.

Proposition 4 *Let $\mathcal{P}$ be a collection of rooted semi-labelled trees, and suppose that we apply* AncestralBuild *to $\mathcal{P}$ but with the restricted descendancy graph replacing the descendancy graph. Then the resulting algorithm applied to $\mathcal{P}$ returns either*

(i) *a rooted semi-labelled tree that ancestrally displays $\mathcal{P}$ if $\mathcal{P}$ is ancestrally compatible, or*

(ii) *the statement $\mathcal{P}$* is not ancestrally compatible *otherwise.*

The statement of Proposition 4 is the same statement as Theorem 4.1 (Daniel and Semple, 2004), but without the proviso on using the restricted descendancy graph. Thus, not surprisingly, the proof of Proposition 4 is very similar to their proof. Consequently, to avoid repetition, we refer to parts of the latter proof where appropriate.

Proof of Proposition 4. We first note that, as in the proof of Theorem 4.1 (Daniel and Semple, 2004), it suffices to show that the result holds when $\mathcal{P}$ is a collection of rooted fully-labelled trees. The proof of (i) is the same as the proof of Theorem 4.1 (Daniel and Semple, 2004).

For the proof of (ii), suppose that ANCESTRALBUILD (using the restricted descendancy graph) outputs a rooted semi-labelled tree $\mathcal{T}'$. We show that $\mathcal{T}'$ ancestrally displays $\mathcal{P}$. Let $\mathcal{T}_1$ be an element of $\mathcal{P}$. By Lemma 2.1 (Bordewich et al., 2005), it suffices to show, for all $a, b \in \mathcal{L}(\mathcal{T}_1)$ that (I) if a is a descendant label of b in $\mathcal{T}_1$, then a is a descendant label of b in $\mathcal{T}'$, and (II) if a and b are non-comparable in $\mathcal{T}_1$, then a and b are non-comparable in $\mathcal{T}'$.

The argument for (I) is very similar to the corresponding argument in the proof of Theorem 4.1(ii) (Daniel and Semple, 2004), and so we omit it. To show (II), suppose that a and b are not comparable in $\mathcal{T}_1$. Assume that $\mathcal{T}_1 = (T_1; \phi_1)$. Let v be the node in T_1 that is the most recent common ancestor of $\phi_1(a)$ and $\phi_1(b)$. By the construction of $D^*(\mathcal{P})$, there is a pair of children, c and d say, of the label labelling v in T_1 such that c and d are joined by an edge, and c is an ancestor label of a, and d is an ancestor label of b. Since we eventually output a tree, this edge is eventually deleted, but not until c and d, and hence a and b, are in separate arc components of some restriction of $D^*(\mathcal{P})$. It now follows that a and b are not comparable in $\mathcal{T}'$. □

Remark. Let $\mathcal{P}$ be a collection of rooted semi-labelled trees with $|\mathcal{P}| = t$ and $|\mathcal{L}(\mathcal{P})| = n$, and let $\mathcal{P}'$ be a collection of fully-labelled trees that is obtained from $\mathcal{P}$ by adding distinct new labels. Let $m = \sum_{\mathcal{T} \in \mathcal{P}} |\mathcal{T}|$. Then the mixed graph $D^*(\mathcal{P}')$ contains $O(m)$ nodes and arcs. However, the number e of edges in $D^*(\mathcal{P}')$ is a function of the degree of the nodes in the source trees. In particular, $D^*(\mathcal{P}')$ contains

$$O\Big(\sum_{\mathcal{T}_i \in \mathcal{P}} \sum_{u \in I(T_i)} d(u)^2\Big)$$

edges, where $I(T_i)$ denotes the set of interior nodes of tree T_i for all i. Note that, depending on the degree of overlap of the source trees and the degree of each of their nodes, e can range from $O(m)$ to $O(tn^2)$. In particular, if the source trees are all fully-resolved, then $D^*(\mathcal{P}')$ contains $O(m)$ nodes, arcs, and edges.

Computing the arc components via a smaller graph

Despite the obvious improvements given by Proposition 4, it is Step 3(b) (finding the arc components) and to a lesser extent Step 3(c) (finding the edges joining distinct arc components) that have the biggest influence on the running time of ANCESTRALBUILD because these steps are performed a high number of times on relatively large graphs. To speed-up these parts of the algorithm, we introduce an additional graph (which we call the "component graph"). The reason for this graph is that identifying the arc components of the restricted descendancy graph can be reduced to identifying the connected components in the component graph. The component graph is typically smaller than the restricted descendancy graph, and it is this smallness that provides the improvement in the running time of the algorithm. To describe the component graph, we first need to some additional concepts.

Let $\mathcal{T} = (T; \phi)$ be a rooted semi-labelled tree on X, where T has node set V. We say that $\mathcal{T}'$ is obtained from $\mathcal{T}$ by *adding most recent common ancestor labels* if, for each node z of degree at least three in which $\phi^{-1}(z)$ is empty, we assign the label $\mathrm{mrca}_{\mathcal{T}'}(a, b)$ to z, where $a, b \in \mathcal{L}(\mathcal{T})$ and z is the most recent common ancestor of $\phi(a)$ and $\phi(b)$. By choosing leaf labels if necessary, we can always find appropriate choices for a and b. Since each of the newly added labels are distinct, this construction is a special case of adding distinct new labels. In the paper, we often view the label $\mathrm{mrca}_{\mathcal{T}'}(a, b)$ as the set $\{a, b\}$ and freely move between the two viewpoints. Note that the choice of a and b need not be unique, however, which choice is made is irrelevant. In general, if $\mathcal{P}$ is a collection of rooted semi-labelled trees, we say that $\mathcal{P}'$ has been obtained from $\mathcal{P}$ by *adding most recent common ancestor labels* if it has been obtained by adding most recent common ancestor labels to each tree in $\mathcal{P}$. Note that the newly added labels across $\mathcal{P}'$ are distinct as every most recent common ancestor label refers to a particular tree in $\mathcal{P}$.

Example 5 To illustrate the last construction, the collection $\mathcal{P}'$ of rooted semi-labelled trees shown in Figure 3 has been obtained from the collection $\mathcal{P}$ of rooted semi-labelled trees shown in Figure 1 by adding most recent common ancestors labels. For instance, the label $\mathrm{mrca}_{\mathcal{T}_1}(x, y)$ is assigned to the node of $\mathcal{T}_1$ that is the most recent common ancestor of nodes labelled x and y. □

Let $\mathcal{P}$ be a collection of rooted semi-labelled trees on X, and let $\mathcal{P}'$ be a collection of rooted fully-labelled trees obtained from $\mathcal{P}$ by adding most recent common ancestor labels. To describe the component graph of $\mathcal{P}'$, we simultaneously consider the restricted descendancy graph of $\mathcal{P}'$. For a tree T and a node z in T, we say that u and v are *siblings* if both u and v are distinct children of z. Let $\mathcal{T} = (T; \phi)$ be an element of $\mathcal{P}'$, and let u and v be siblings of a node z in T, and consider $\phi^{-1}(u)$ and $\phi^{-1}(v)$. By the definition of the restricted descendancy graph of $\mathcal{P}'$, we have that $\phi^{-1}(u)$ and $\phi^{-1}(v)$ are joined by an edge e in $D^*(\mathcal{P}')$. (Note that all edges of $D^*(\mathcal{P}')$ are derived from a tree in $\mathcal{P}'$ in this way.) Furthermore, in $D^*(\mathcal{P}')$, there is an arc from $\phi^{-1}(z)$ to $\phi^{-1}(u)$ and an arc from $\phi^{-1}(z)$ to $\phi^{-1}(v)$. Relative to $\mathcal{T}$, we call the set

$$\big\{\{a, b\} : a \in \phi^{-1}(u) \text{ and } b \in \phi^{-1}(v)\big\}$$

the *sibling edge set of* e or, if we are referring to $\phi^{-1}(z)$, we call it a *sibling edge set of* $\phi^{-1}(z)$. Moreover, $\phi^{-1}(z)$ is the *parent node* of this edge set.

The *component graph of* $\mathcal{P}'$, denoted $C(\mathcal{P}')$, has node set X and two types of edges. In particular, for two nodes a and b, there is

(i) an unlabelled edge joining a and b precisely if there is a tree $\mathcal{T} \in \mathcal{P}$ with b an ancestor of a and no element x in $\mathcal{L}(\mathcal{T}) - \{a, b\}$ such that b is an ancestor of x and x is an ancestor of a; and

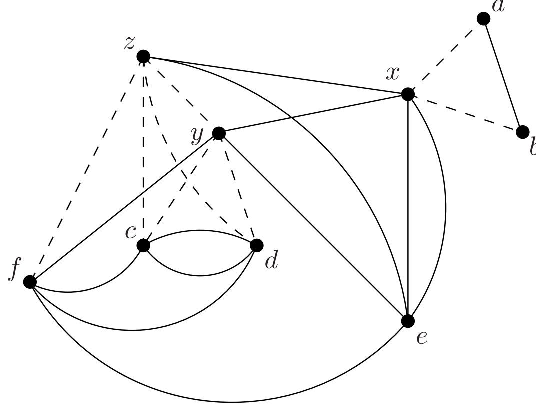

Figure 7: The component graph of $\mathcal{P}'$.

(ii) an edge joining a and b labelled (ℓ, e) precisely if, relative to a tree $\mathcal{T} \in \mathcal{P}'$ with label ℓ, we have that $\{a, b\}$ is in a sibling edge set of ℓ and an edge e in $D^*(\mathcal{P}')$.

Edges in $C(\mathcal{P}')$ arising because of (i) are called *type (i) edges* and edges arising because of (ii) are called *type (ii) edges*. Note that type (i) edges are decided by the composition of $\mathcal{P}$, while type (ii) edges are decided by that of $\mathcal{P}'$. Also, two nodes in $C(\mathcal{P}')$ can be joined by more than one edge. However, these edges are not treated equally as there is at most one unlabelled edge and each of the labelled edges are labelled with different ordered pairs.

Example 6 Figure 7 shows the component graph $C(\mathcal{P}')$ for the collection $\mathcal{P}'$ of rooted fully-labelled trees shown in Figure 3, where type (i) edges are represented as dashed edges and type (ii) edges are represented as solid edges. For clarity, type (ii) edges are not labelled. To illustrate, $\{z, y\}$ is a type (i) edge resulting from the fact that z is an ancestor of y in the tree $\mathcal{T}_2'$ of $\mathcal{P}'$; the fact that $\{z, e, x\}$ are siblings in this tree results in the three type (ii) edges $\{z, e\}$, $\{z, x\}$, and $\{e, x\}$ in $C(\mathcal{P}')$. Viewing $\mathrm{mrca}_{\mathcal{T}_1}(e, y)$ as $\{e, y\}$, the second edge $\{e, x\}$ joining e and x results from the fact that $\mathrm{mrca}_{\mathcal{T}_1}(e, y)$ and x are siblings in $\mathcal{T}_1'$, which generates the sibling edge set $\big\{\{e, x\}, \{y, x\}\big\}$ in $C(\mathcal{P}')$. Thus one of the edges joining e and x is labelled $(\mathrm{mrca}_{\mathcal{T}_2}(e, z), \{e, x\})$, while the other edge joining e and x is labelled $(\mathrm{mrca}_{\mathcal{T}_1}(x, y), \{\mathrm{mrca}_{\mathcal{T}_1}(e, y), x\})$. □

Remark. Asymptotically, the component graph $C(\mathcal{P}')$ contains the same number of edges as $D^*(\mathcal{P}')$. However, $C(\mathcal{P}')$ contains only n nodes, whereas $D^*(\mathcal{P}')$ contains $O(m)$ nodes. Potentially, this means that the latter gains a factor as the size of m is $O(tn)$. Thus, computing connected components in $C(\mathcal{P}')$ is likely to be faster than computing them in $D^*(\mathcal{P}')$. Indeed, in the next section we show how the computation of arc components in Step 3(b) and determining which edges are to be deleted in Step 3(c) can be made faster by resorting to $C(\mathcal{P}')$.

At last we describe our full modification of AncestralBuild called Ancestral-Build*. Analogous to Descendant in AncestralBuild, the algorithm Ancestral-

Build* includes a subroutine which we call Descendant*. Intuitively, apart from using the restricted descendancy graph instead of the descendancy graph, the main difference is that all of the work in finding the arc components and deciding which edges join two different arc components at each iteration of the subroutine is now done by the component graph and its various node induced subgraphs. In the modification, all of the edges of the component graph are initially coloured blue. In association with the component graph (or any of its node induced subgraphs), a *blue component* is a connected component of the graph obtained when masking non-blue edges. To describe AncestralBuild*, we highlight the changes to AncestralBuild:

(i) In Step 1, $\mathcal{P}'$ is now obtained from $\mathcal{P}$ by adding most recent common ancestor labels.

(ii) Step 2 is replaced with the constructions of the restricted descendancy graph $D^*(\mathcal{P}')$ and the component graph $C(\mathcal{P}')$ of $\mathcal{P}'$.

(iii) The input to the subroutine Descendant is initially $D^*(\mathcal{P})$ and $C(\mathcal{P}')$. For recursive calls to Descendant (Step 4), the input is $D^*(\mathcal{P}')|\mathcal{S}_i$ and $C(\mathcal{P}')|(\mathcal{S}_i \cap X)$.

(iv) Step 3 of Descendant is replaced with Step 3′ (see boxed insert).

3′. Otherwise,

(a) **(i)** Delete the elements of $\mathcal{S}_0$ (and their incident arcs) from $D^*(\mathcal{P}')$ and denote the resulting graph by $D^*(\mathcal{P}')\backslash\mathcal{S}_0$.

(ii) Delete the elements of $\mathcal{S}_0 \cap X$ (and their incident edges) from $C(\mathcal{P}')$ and denote the resulting graph by $C(\mathcal{P}')\backslash(\mathcal{S}_0 \cap X)$.

(iii) For every element of $\mathcal{S}_0$, colour all edges in each of its sibling edge sets in $C(\mathcal{P}')\backslash(\mathcal{S}_0\cap X)$ red.

(b) Find the node sets $\mathcal{U}_1, \mathcal{U}_2, \ldots, \mathcal{U}_k$ of the blue components of $C(\mathcal{P}')\backslash(\mathcal{S}_0 \cap X)$. The arc components of $D^*(\mathcal{P}')\backslash\mathcal{S}_0$ are $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k$, where $\mathcal{S}_i \cap X = \mathcal{U}_i$ for all i.

(c) Delete all red edges joining two different blue components of $C(\mathcal{P}')\backslash(\mathcal{S}_0)$. For each sibling edge set that is deleted, delete the corresponding edge in $D^*(\mathcal{P}')\backslash\mathcal{S}_0$.

The modified subroutine is called Descendant*.

Remarks. In the following remarks and in the next section, we will assume for reasons of convenience that the input to Descendant* is always $D^*(\mathcal{P}')$ and $C(\mathcal{P}')$. This is in name only. (Strictly speaking, the input of recursive calls are node induced subgraphs of these graphs.)

1. At the beginning and at the end of each iteration of Descendant*, the node sets of the blue components of $C(\mathcal{P}')$ correspond to the node sets of the arc components of $D^*(\mathcal{P}')$.

2. Although deleting the elements in $\mathcal{S}_0$ and their incident edges in $D^*(\mathcal{P}')$ has the potential to create arc components $A_1, A_2, \ldots, A_k$, deleting the elements in $\mathcal{S}_0 \cap X$ in $C(\mathcal{P}')$ will not create any blue components. This is because the sibling edges corresponding to the elements in $\mathcal{S}_0$ are still coloured blue in the resulting subgraph of $C(\mathcal{P}')$. However, Step 3′(a)(iii) colours these sibling edges red and it is this recolouring which reestablishes the correspondence described in Step 3′(b).

3. The fact that the arc components of $D^*(\mathcal{P})$ correspond to the blue components of $C(\mathcal{P}')\backslash(\mathcal{S}_0 \cap X)$ as stated in Step 3′(b) is established in Lemma 12.

4. Referring to Step 3′(c) of Descendant*, the set of red edges that are deleted is a union of sibling edge sets. Again this is established in Lemma 12.

Example 7 Consider the execution of AncestralBuild* on the collection of rooted semi-labelled trees shown in Figure 1 which in Step 1 constructs the collection $\mathcal{P}'$ of rooted fully-labelled trees shown in Figure 3 by adding most recent common ancestors. Refer to Figures 4 and 7, respectively, for $D^*(\mathcal{P}')$ and $C(\mathcal{P}')$ which are constructed in Step 2.

At the first call of Descendant*, Step 3′(a)(i) deletes the nodes in

$$\mathcal{S}_0 = \{\mathrm{mrca}_{\mathcal{T}_1}(x, y), \mathrm{mrca}_{\mathcal{T}_2}(e, z)\}$$

from $D^*(\mathcal{P}')$. These nodes are not in the original taxa set X, thus Step 3′(a)(ii) has nothing to delete from $C(\mathcal{P}')$. Step 3′(a)(iii) colours the edges in the sibling edge sets of $\mathrm{mrca}_{\mathcal{T}_1}(x, y)$ and $\mathrm{mrca}_{\mathcal{T}_2}(e, z)$ red in $C(\mathcal{P}')$; that is, the edges in $\big\{\{e, x\}, \{y, x\}\big\}$ and $\big\{\{z, e\}, \{z, x\}, \{e, x\}\big\}$, respectively, where we recall that the edge $\{e, x\}$ in each of these sets is treated differently; one is labelled $(\mathrm{mrca}_{\mathcal{T}_1}(x, y), \{\mathrm{mrca}_{\mathcal{T}_1}(e, y), x\})$ and the other is labelled $(\mathrm{mrca}_{\mathcal{T}_2}(e, z), \{e, x\})$ in $C(\mathcal{P}')$. As a result, Step 3′(b) identifies two blue components, $\mathcal{U}_1$ and $\mathcal{U}_2$ say, with a, b, and x in $\mathcal{U}_1$ and the remaining nodes in $\mathcal{U}_2$. The corresponding arc components of $D^*(\mathcal{P}')\backslash\mathcal{S}_0$ are the same as the ones found in Example 2 during the first call to Descendant. Step 3′(c) removes from $C(\mathcal{P}')$ the red edges between $\mathcal{U}_1$ and $\mathcal{U}_2$; that is, the edges in $\big\{\{e, x\}, \{y, x\}, \{z, x\}, \{e, x\}\big\}$, leaving $\{z, e\}$ as the only red edge of the graph at this point. The rooted semi-labelled tree that is eventually returned at the end of recursive calls is the tree shown in Figure 2. The fact that this is the same tree as the one outputted by AncestralBuild applied to $\mathcal{P}$ is no coincidence (see Theorem 8). □

Together with Proposition 4, the correctness of AncestralBuild* is established in Appendix . In particular, we have the following theorem.

Theorem 8 *Let $\mathcal{P}$ be a collection of rooted semi-labelled trees. Then, applying the algorithm* AncestralBuild* *to $\mathcal{P}$ returns either*

(i) *a rooted semi-labelled tree that ancestrally displays $\mathcal{P}$ if $\mathcal{P}$ is ancestrally compatible, or*

(ii) *the statement* $\mathcal{P}$ is not ancestrally compatible *otherwise.*

Moreover, if AncestralBuild* *returns a tree, then (up to isomorphism) the tree returned by* AncestralBuild* *is the same as that returned by* AncestralBuild *when applied to $\mathcal{P}$.*

Using a dynamic data structure to reduce the computation time of connected components

Different calls to the subroutine Descendant* consider different restrictions of the initial component graph $C(\mathcal{P}')$. Moreover, the recursive calls issued during an on-going call of the subroutine consider node disjoint restrictions of $C(\mathcal{P}')$. This shows that $C(\mathcal{P}')$ can be shared by all calls of the subroutine without the need to keep copies of parts of it. Edges are progressively removed from $C(\mathcal{P}')$ as the different calls to Descendant* are executed. Each call has to determine the resulting blue connected components of the part of $C(\mathcal{P}')$ it is considering.

As shown in Henzinger et al. (1999) for a similar problem, this problem greatly simplifies if connected components of the graph are maintained in a separate data structure handled by a dynamic connectivity algorithm. This ad-hoc data structure ensures that connectivity queries on the graph (*i.e.*, asking whether two given nodes are in the same component) and updates of the graph (here, removing an edge) are performed efficiently. For example, the dynamic algorithm of Holm et al. (1998) supports each connectivity query in $O(\log n/ \log\log n)$ and each edge deletion update in $O(\log^2 n)$, where n is the number of nodes of the graph. Many implementations of dynamic connectivity algorithms have been proposed, and we refer the reader to Zaroliagis (2002) for a recent survey and experimental comparison of their running times. Next section details how the dynamic data structure comes into play in the execution of Steps $3'$(b) and $3'$(c) of Descendant*.

Complexity of AncestralBuild*

In this section, we establish the running time of AncestralBuild*. In addition to the implementation details given here, more concrete details can be found in Appendix .

For a collection $\mathcal{P}$ of rooted semi-labelled trees, the burden of the computation in AncestralBuild*($\mathcal{P}$) lies in the subroutine Descendant* and, more particularly, in Steps $3'$(b) and $3'$(c) when dealing with the computations in the graph $C(\mathcal{P}')$. For the

exposition that follows, let n be the number of nodes and e' be the number of edges of $C(\mathcal{P}')$.

Step $3'$(b) identifies the blue components of $C(\mathcal{P}')\backslash(\mathcal{S}_0 \cap X)$. At the beginning of a call to DESCENDANT*, there is only one blue component, and then new ones result from the deletion of nodes and the change of colour of edges performed in Step $3'$(a) of the on-going call. As already remarked, these new blue components arise more precisely at Step $3'$(a)(iii). To determine resulting new blue components, Step $3'$(b) sends one by one deletion updates to the dynamic algorithm for each of the edges $\{a, b\}$ turned red in Step $3'$(a)(iii). After each such deletion update, a connectivity query is issued to the dynamic connectivity algorithm to check whether a and b are still in the same component. If the answer is negative, then turning red this edge resulted in the division of a blue component C in $C(\mathcal{P}')$ into two blue components, C_a and C_b say, where a is in the node set of C_a and b is in the node set of C_b. Starting with a and b, the other nodes of these components are then discovered by examining (via blue edges) neighbors of nodes that are in C_a and C_b, respectively. This examination processes each new node alternatively for C_a and C_b, and halts as soon as all nodes of the smallest of the two components have been found. This small component is considered *new* and the other is considered to be the original component C having lost some nodes. This technique, due to Even and Shiloach (1981), guarantees that each of the e' edges in the graph belongs to a new component at most $\log n$ times over all executions of Step $3'$(b). This bounds the number of times an edge is examined whilst it is blue. Moreover, the overall number of deletion updates and connectivity queries issued to the dynamic algorithm is proportional to the number of edges initially in the graph, *i.e.*, $O(e')$.

Step $3'$(c) identifies and deletes all red edges joining two distinct blue components of $C(\mathcal{P}')\backslash(\mathcal{S}_0 \cap X)$. This is done by examining red edges separating *new* (hence small) blue components from other blue components. To identify these edges, all red edges incident to a node in a new blue component are examined. Those edges that are not separating two blue components are ignored (they will be removed at a later stage); those separating two blue components are removed from $C(\mathcal{P}')\backslash(\mathcal{S}_0 \cap X)$. Note that it is possible to distinguish between these two situations in constant time without issuing a connectivity query to the dynamic algorithm. It suffices to associate a number to each blue component and to maintain in Step $3'$(b) a table indicating for each node of $C(\mathcal{P}')$ the number of the blue component to which it currently belongs. Since new blue components are small (*i.e.*, contain at most half of the nodes of the component from which they originate), each red edge is examined at most $\log n$ times before being deleted from the graph. Due to Henzinger et al. (1999), this technique leads to an overall running time of $O(e' \log n)$ to delete red edges between blue components over all executions of Step $3'$(c).

Lemma 9 *Let $\mathcal{P}$ be a collection of rooted semi-labelled trees with $|\mathcal{L}(\mathcal{P})| = n$. Performing Steps $3'$(b) and $3'$(c) over all executions of* DESCENDANT*$(D^*(\mathcal{P}'),\ C(\mathcal{P}'))$ *during algorithm* ANCESTRALBUILD*$(\mathcal{P})$ *costs $O(e \log^2 n)$ running time, where e is the initial number of edges in $D^*(\mathcal{P}')$.*

Proof. Let e' be the initial number of edges in $C(\mathcal{P}')$. As stated above, computing the blue components of $C(\mathcal{P}')$ over all executions of Step 3′(b) necessitates $O(e' \log n)$ operations on the graph $C(\mathcal{P}')$ plus $O(e')$ deletion updates and connectivity queries to a dynamic algorithm maintaining connectivity between nodes in $C(\mathcal{P}')$ through an ad-hoc data structure. Using the dynamic algorithm of Holm et al. (1998) leads to an $O(e' \log^2 n)$ running time for all executions of this step.

As also stated above, each of the e' edges of $C(\mathcal{P}')$ is investigated at most $\log n$ times during all executions of Step 3(c) with each such investigation being done in constant time. Thus, removing red edges between blue components globally requires $O(e' \log n)$ time. Now edges of $D^*(\mathcal{P}')$ that correspond to these red edges of $C(\mathcal{P}')$ are known immediately because of pointers which are maintained between each edge of $D^*(\mathcal{P}')$ and its associated sibling edge set in $C(\mathcal{P}')$. Thus, when all edges in a sibling edge set have been removed from $C(\mathcal{P}')$, removing the corresponding edge in $D^*(\mathcal{P}')$ is done in constant time. As there are $O(e)$ edges in $D^*(\mathcal{P}')$, this requires $O(e)$ time over the whole execution of AncestralBuild*.

Hence, Step 3′(b) is the most time consuming, and noting that $e' = O(e)$ gives the stated result. □

Since Steps 3′(b) and 3′(c) of the subroutine Descendant* are the most time consuming steps during an execution of AncestralBuild*, Theorem 10 is an immediate consequence of Lemma 9.

Theorem 10 *Let $\mathcal{P}$ be a collection of rooted semi-labelled trees with $|\mathcal{L}(\mathcal{P})| = n$. Then* AncestralBuild*$(\mathcal{P})$ *runs in time*

$$O\big(\log^2 n \cdot \big(\sum_{T_i \in \mathcal{P}} \sum_{u \in I(T_i)} d(u)^2\big)\big),$$

where $I(T_i)$ denotes the set of interior nodes of T_i for all i.

Remark. In general, AncestralBuild* allows for the source trees to be rooted semi-labelled trees of unbounded degree. However, in the special case where the source trees are all rooted binary semi-labelled trees, the running time of AncestralBuild* is $O(m \log^2 n)$. This running time is the same as the running time of the algorithm in Henzinger et al. (1999) whose source trees are all rooted binary phylogenetic trees. (The running time of this algorithm can be improved from $O(mn^{\frac{1}{2}})$ (as stated in Henzinger et al. (1999)) to $O(m \log^2 n)$ by changing the dynamic connectivity algorithm it resorts to.) This running time is almost linear in the size of the input which guarantees short execution times even on large data sets.

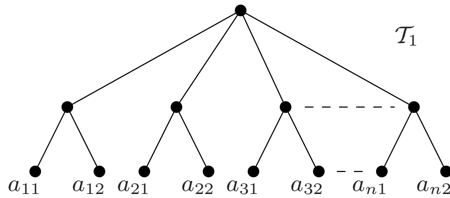

Figure 8: The rooted phylogenetic tree $\mathcal{T}_1$.

A comparison of running times for partially-resolved trees

Let $\mathcal{P}$ be a collection of rooted semi-labelled trees. Ideally, one would like the running time of an algorithm that determines the compatibility of $\mathcal{P}$ to not depend on whether or not $\mathcal{P}$ contains partially-resolved trees. The method ANCESTRALBUILD* has this dependency; the running time in Theorem 10 includes the factor $\sum_{\mathcal{T}_i \in \mathcal{P}} \sum_{u \in I(T_i)} d(u)^2$. The reason for this factor is that if $\mathcal{P}'$ is a collection of rooted fully-labelled trees obtained from $\mathcal{P}$ by adding most recent common ancestor labels, then, for each tree $\mathcal{T}'$ in $\mathcal{P}'$ and each label ℓ in $\mathcal{L}(\mathcal{T}')$, the number of edges in the descendancy graph of $\mathcal{P}'$ joining pairs of siblings of ℓ is quadratic in the number of siblings. Unfortunately, given our current approach, there appears to be no way to remedy this. To see this, suppose that one can always choose a linear number of such edges. We will assume that this choice is independent amongst the trees in $\mathcal{P}'$. In the consideration of running times, this assumption is reasonable, for otherwise, one has to make $O(t^2)$ comparisons amongst the trees in $\mathcal{P}'$, where $|\mathcal{L}(\mathcal{P})| = t$. We next describe a collection of rooted semi-labelled trees such that using only a linear number of edges in $D^*(\mathcal{P}')$ leads ANCESTRALBUILD* to incorrectly return a tree when applied to this collection.

A *rooted triple* is a rooted phylogenetic tree that has two interior nodes and whose label set has size three. We denote the rooted triple $\mathcal{T}$ with label set $\{a, b, c\}$ by $ab|c$ if the path from a to b does not intersect the path from the root to c.

Let $\mathcal{P} = \{\mathcal{T}_1, \mathcal{T}_2\}$, where $\mathcal{T}_1$ is the rooted phylogenetic tree shown in Figure 8 and $\mathcal{T}_2$ is a rooted triple that will be described shortly. Suppose that ANCESTRALBUILD* is applied to $\mathcal{P}$ with the linearity condition described above. Let $\mathcal{P}'$ be the collection of rooted fully-labelled trees obtained in Step 1 of ANCESTRALBUILD*. Because of our assumption, the number of pairs of elements of

$$\big\{\{a_{11}, a_{12}\}, \{a_{21}, a_{22}\}, \ldots, \{a_{n1}, a_{n2}\}\big\}$$

which are joined by edges in the descendancy graph $D(\mathcal{P}')$ of $\mathcal{P}'$ is linear in n. This implies that there is a pair, $\{a_{11}, a_{12}\}$ and $\{a_{21}, a_{22}\}$ say, not joined by an edge. Now set $\mathcal{T}_2$ to be the rooted triple $a_{12}a_{21}|a_{22}$. Clearly, $\mathcal{T}_1$ and $\mathcal{T}_2$ are not compatible, yet the rooted semi-labelled tree shown in Figure 9 is returned by this application of ANCESTRALBUILD*.

The running-time dependency on partially-resolved trees may possibly be inherent in any supertree method for determining compatibility of the input collection, even in the

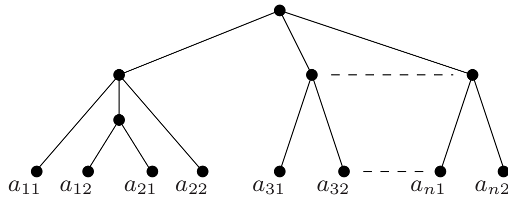

Figure 9: The tree outputted by AncestralBuild* when applied to $\{\mathcal{T}_1, \mathcal{T}_2\}$ with a particular linearity condition.

case this collection consists of just leaf-labelled trees. To make a comparison with the running time in Theorem 10, we examine what appears to be the most canonical and natural extension of the algorithm in Henzinger et al. (1999) for deciding the compatibility of a collection $\mathcal{P}$ of rooted binary phylogenetic trees to deciding the compatibility of a collection of arbitrary rooted phylogenetic trees.

To aid the running time of the algorithm in Henzinger et al. (1999), the source trees in $\mathcal{P}$ are encoded as a collection of rooted triples. The resulting collection is displayed by a rooted phylogenetic tree $\mathcal{T}$ if and only if $\mathcal{P}$ is displayed by $\mathcal{T}$. Extending this to a collection of source trees that contain arbitrary rooted phylogenetic trees, the minimum number of rooted triples required for the encoding is $O(\sum_{T_i \in \mathcal{P}} \sum_{u \in I(T_i)} d(u)^2)$ (Grünewald et al., 2005). The complexity of the resulting algorithm would be the same as the one stated in Theorem 10 for AncestralBuild*.

An Example on Primates

As an application of AncestralBuild*, we now consider the phylogeny of Strepsirrhini, one of the two major groups of primates. To infer this phylogeny, we use the ability of supertree methods to indirectly combine data of different kinds and AncestralBuild* to combine a set of source trees with nested taxa. The supertree is obtained from four source trees deriving from (i) retroposon data (Roos et al., 2004), (ii) morphological data (Masters and Brothers, 2002), and (iii) molecular data (Yoder et al., 2000) (see Figure 10).

The first source tree (a) has been obtained from retroposon analyses (Roos et al., 2004, Fig. 2) namely from 61 loci containing short interspersed elements (SINEs), translated into a presence-absence pattern at orthologous loci on 21 strepsirrhine species. This data contains no homoplasy and indicates unambiguously a unique tree. However, this tree does not resolve all phylogenetic relationships of the strepsirrhines: it exhibits a trifurcation involving *Galagoides, Otolemur*, and *Galago*, as well as a second trifurcation involving the three groups of the Lemuroidea (Lepilemur, Cheirogaleidae, and the group composed of Lemuridae and Indridae).

The second source tree (b) contains 13 species of the Galagonidae family and has been

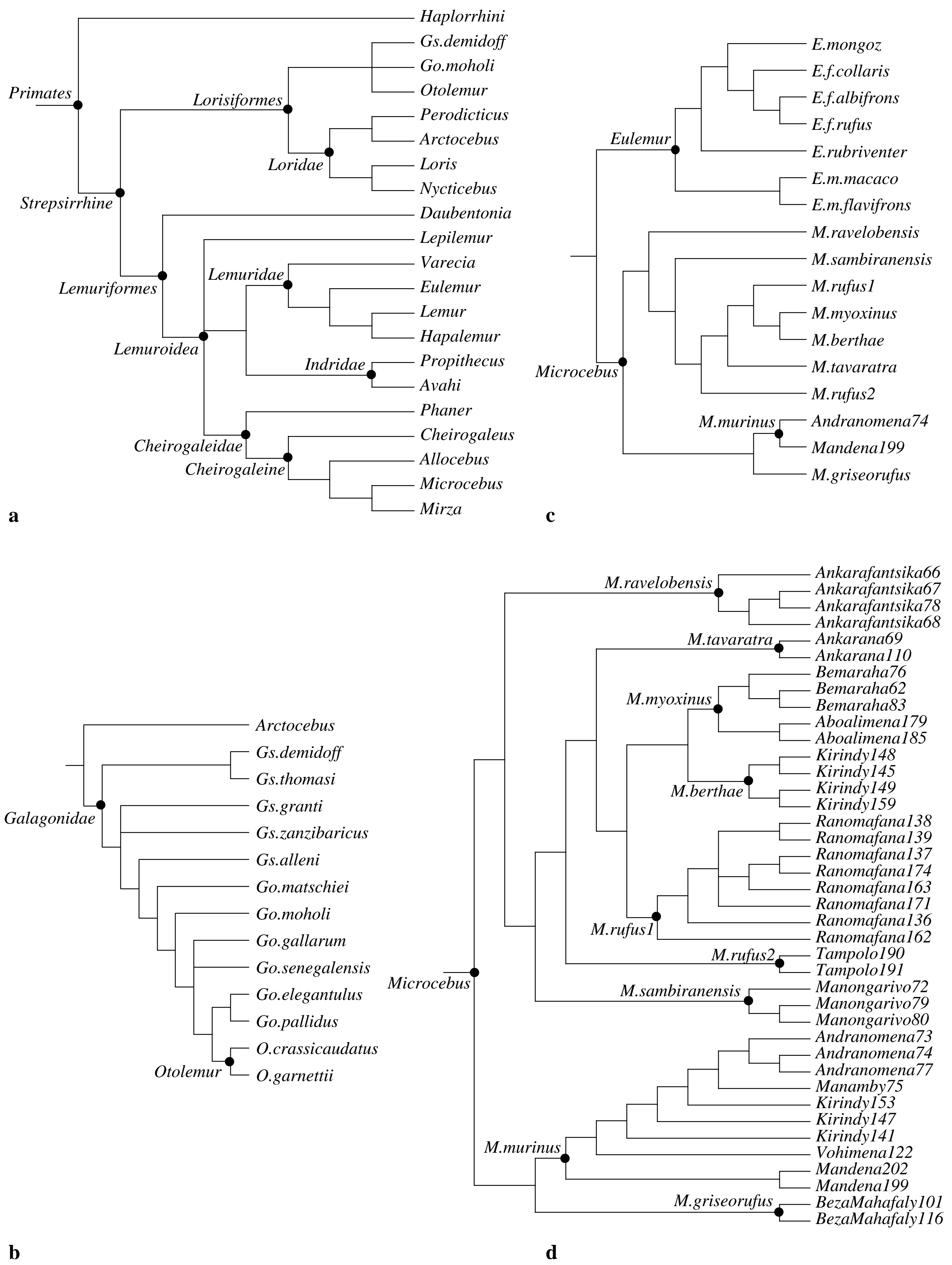

Figure 10: Four source trees of Strepsirrhini, one of the two major groups of primates. These source trees are derived from retroposon, morphological, and molecular data.

inferred from craniodental morphological data (Masters and Brothers, 2002, Fig. 6.a). This tree resolves the first trifurcation mentioned above. As it is the strict consensus of the two most parsimonious trees, this tree also contains multifurcations. More precisely, the two observed trifurcations respectively concern the placement of two *Galagoïdes* species and of two *Galago* species.

The third and fourth source trees (c) and (d) have been inferred from mtDNA sequences combined from the control region homologous with the hypervariable region 1 in humans, COII and cytochrome b (Yoder et al., 2000, Fig. 2 and Fig. 3). The third tree (c) contains 18 species and subspecies of *Microcebus* and *Eulemur*. The fourth tree (d) contains 40 individuals of the *Microcebus* genus arranged in 9 identified species.

Internal labels of trees (b), (c), and (d) correspond to those displayed in the original figures, while those in tree (a) were added manually to demonstrate the ability of AncestralBuild* to deal accurately with many labels, located at the same or different levels in the trees. The four detailed source trees are ancestrally compatible and Figure 11 shows the supertree resulting from the application of AncestralBuild* to this collection. The obtained phylogeny is one of the largest produced for the strepsirrhines, spanning approximately 100 taxa on a number of taxonomic levels, from order to individuals. The source trees used in this example as well as the final supertree are available online from `http://www.systematicbiology.org`.

Discussion

AncestralBuild* does not take primary data as input, but rather source trees inferred from this data with some level of confidence and through an adequate method. Thus, it is likely that the source trees considered for building a supertree will be more often compatible than, say, a set of primary character data. Nonetheless, it is likely that in many cases the source trees turn out to be incompatible. Providing a faster way than other current supertree methods to detect this incompatibility is a first goal of the algorithm presented in this paper. However, this does not mark the end of its use in the process of building a supertree. Indeed,

(i) AncestralBuild* can be integrated in a general supertree method that builds a supertree by resolving incompatibilities in the source trees.

(ii) Moreover, AncestralBuild* as a whole can be used repeatedly to quickly identify compatible subsets of the set of source trees or parts of the source trees that are compatible. The resulting compatible subsets or parts are then combined into a supertree using AncestralBuild*.

We indicate below some hints in both these directions.

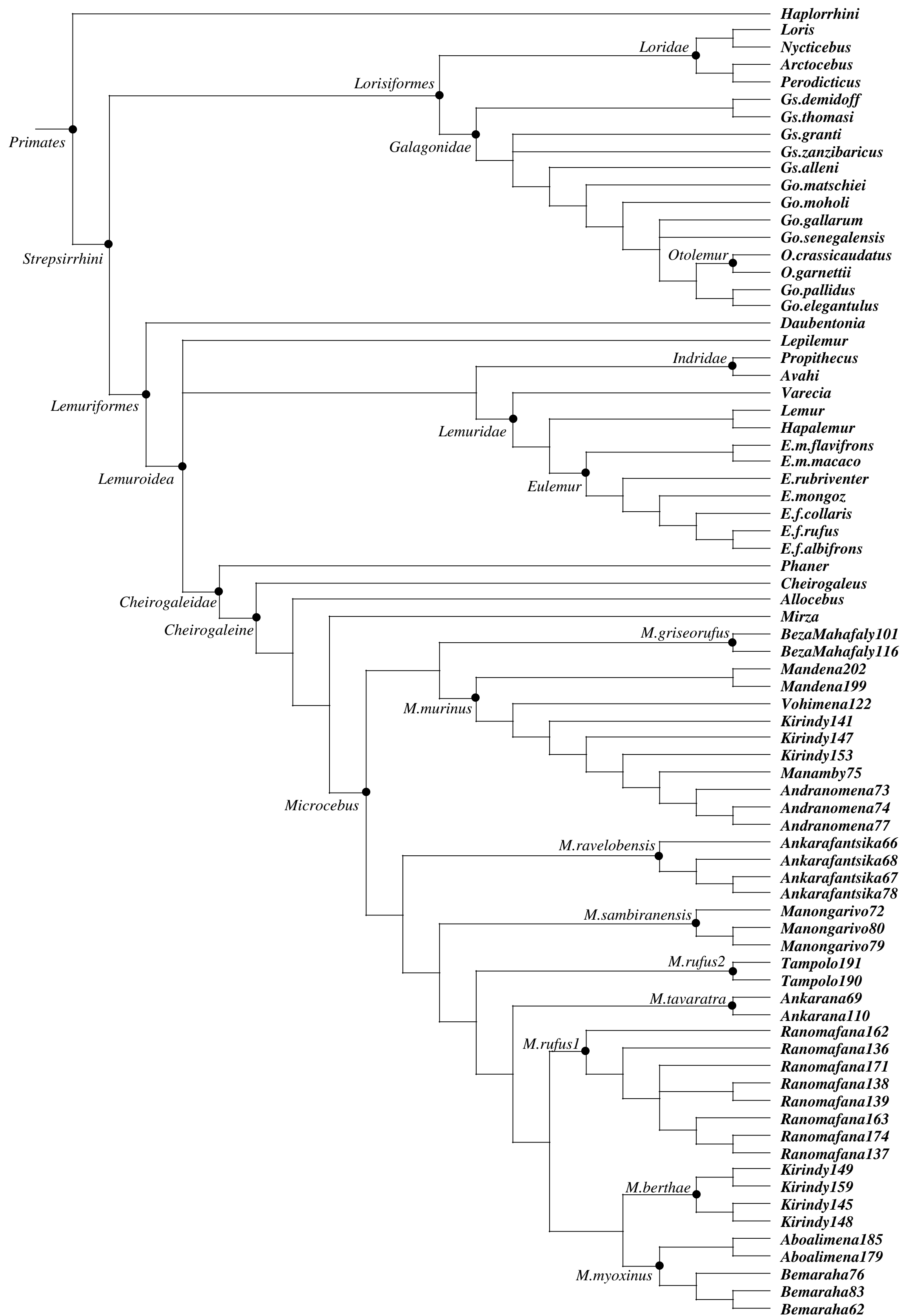

Figure 11: Supertree of the strepsirrhines resulting from four compatible source trees, inferred from retroposon, morphological, and molecular data. This phylogeny demonstrates the ability of AncestralBuild* to deal with rooted semi-labelled trees spanning many different taxonomic levels, from individuals (*e.g.*, 66-Ankarafantsika) to order (*i.e.*, Primates).

Integration of AncestralBuild* in a general supertree method

As stated in the introduction, consistency is an attractive property for any supertree method. Thus, in constructing a general supertree method, deciding compatibility is an integral part of the method. Currently, it seems that the only general supertree methods for rooted semi-labelled trees is given in Daniel and Semple (2005). In this paper, the authors describe a general supertree method that allows for the possibility of variants. This method, called NestedSupertree, extends AncestralBuild, and thus AncestralBuild*. If the source trees are compatible, then it outputs a supertree that ancestrally displays each of these trees. On the other hand, if the source trees are not compatible, then at some iteration there are no nodes that have indegree zero and no incident edges. By making an appropriate choice of nodes to delete, NestedSupertree, or more particularly one of its variants, resolves this and continues on, eventually returning a supertree with several desirable features including the following:

(i) ancestrally displaying every rooted binary semi-labelled trees that is ancestrally displayed by each of the source trees;

(ii) independent of the order in which the source trees are listed.

We also remark that NestedSupertree runs in polynomial time and allows for the source trees to be weighted. Such weights, irrelevant for deciding compatiblity (and thus ignored by AncestralBuild), can really help to arbitrate the conflicts between incompatible source trees.

The progress made in this paper on the running time of AncestralBuild improves the practicality of general supertree methods for nested taxa such as NestedSupertree.

Repeated use of AncestralBuild* in the production of a supertree

Despite the exactness AncestralBuild*, it can still be used to build a supertree from incompatible source trees. Two ways are highlighted below.

- **Finding a subset of the source trees that are compatible.** Given an incompatible collection $\mathcal{P}$ of source trees, finding a maximum-sized subset of trees in $\mathcal{P}$ that are compatible is an NP-hard task (Bryant, 1997). However, heuristic methods can be easily implemented: (i) rank all trees in $\mathcal{P}$ according to their size, or to some confidence value on the trees (*e.g.*, bayesian posterior probabilities) or in the primary data set from which they were obtained; (ii) build a compatible collection $\mathcal{P}' \subseteq \mathcal{P}$ in the following way: starting from the best ranked tree, consider each source tree of $\mathcal{P}$ successively and add it to $\mathcal{P}'$ if it forms a compatible collection with the trees

already in $\mathcal{P}'$, which is checked by AncestralBuild*. At the end of the process, $\mathcal{P}'$ is a subset of compatible source trees, a supertree of which is provided by the final call to AncestralBuild*.

- **Finding parts of the source trees that are compatible.** Usually, source trees result from an extensive analysis of primary data and their clades are provided with associated confidence values, such as bootstrap values or bayesian posterior probabilities. As a first approximation, we may assume that these confidence values are representative in some sense of the correctness of the corresponding clades (see *e.g.* Berry and Gascuel (1996) for a discussion). Thus, when source trees are incompatible, a reasonable option is to first put into question the clades of the source trees that display the least support from the data. This suggests an intuitive and simple scheme to remove conflicts from the source trees by collapsing some of the clades from consideration: Let $\mathcal{O}$ be the list of support values for clades of the source trees, sorted by increasing order of confidence. Note that a clade appearing in different trees with different confidence values can be accounted for by resorting to suitable weighting schemes. Collapse clades of the source trees whose support value is equal to the first value of $\mathcal{O}$ and remove that value from the list. Then iterate until the modified source trees are compatible. Compatibility is checked every time by using AncestralBuild*, the final call providing a supertree from the collection of modified trees.

Acknowledgments

The authors are grateful to E. Douzery and P.-H. Fabre for indicating studies on the strepsirrhine evolution used in this paper and for verifying our findings.

We thank Olaf Bininda-Emonds, Rod Page, Gabriel Valiente, and an anonymous referee for their valuable comments. We expect the idea of Bininda-Emonds regarding a reference taxonomy as additional input to play an important practical role in the use of supertree algorithms.

The first author was supported by the *Act. Incit. Inf.-Math.-Phys. en Biol. Mol.* [ACI IMP-Bio] and the *Act. Inter. Incit. Région.* [BIOSTIC-LR]. The second author was supported by the *New Zealand Marsden Fund* and a *University of Canterbury Erskine Grant*. This work was done while the second author was a Visiting Professor at the Université of Montpellier II.

References

Aho, A. V., Y. Sagiv, T. G. Szymanski, and J. D. Ullman. 1981. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions.

SIAM J. Comput. 10:405–421.

Berry, V. and O. Gascuel. 1996. On the interpretation of bootstrap trees: appropriate threshold of clade selection and induced gain. Mol. Biol. Evol. 13:999–1011.

Bininda-Emonds, O. R. P., ed. 2004. Phylogenetic supertrees: combining information to reveal the Tree of Life. Kluwer, Dordrecht.

Bininda-Emonds, O. R. P., K. E. Jones, S. A. Price, M. Cardillo, R. Grenyer, and A. Purvis. 2004. Garbage in, grabage out. Pages 267–280 *in* Phylogenetic supertrees: combining information to reveal the Tree of Life (O. R. P. Bininda-Emonds, ed.). Kluwer, Dordrecht.

Bordewich, M., G. Evans, and C. Semple. 2005. Extending the limits of supertree methods. Ann. of Comb. (in press) .

Bryant, D. 1997. Building trees, hunting for trees, and comparing trees: theory and methods in phylogenetic analysis. Ph.D. thesis University of Canterbury.

Daniel, P. and C. Semple. 2004. Supertree algorithms for nested taxa. Pages 151–171 *in* Phylogenetic supertrees: combining information to reveal the Tree of Life (O. R. P. Bininda-Emonds, ed.). Kluwer, Dordrecht.

Daniel, P. and C. Semple. 2005. A class of general supertree methods for nested taxa. SIAM J. Discrete Math. (in press) .

Even, S. and Y. Shiloach. 1981. An on-line edge deletion problem. Journal of the Association of Computing Machinery 28:1–4.

Grünewald, S., M. Steel, and M. Swenson. 2005. Closure operations in phylogenetics. Tech. rep. University of Canterbury.

Henzinger, M. R., V. King, and T. Warnow. 1999. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. Algorithmica 24:1–13.

Hibbett, D., R. H. Nilsson, M. Snyder, M. F. andJ Costanzo, and M. Shonfeld. 2005. Automated phylogenetic taxonomy: an example in the homobasidiomycetes (mushroom-forming fungi). Syst.Biol. 54:660–668.

Holm, J., K. de Lichtenberg, and M. Thorup. 1998. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. Pages 78–89 *in* Proceedings of the 30th Annual ACM Symp. on Theory of Computing.

Llabrés, M., J. Rocha, F. Rosselló, and G. Valiente. 2005. On the ancestral compatibility of two phylogenetic trees with nested taxa. Submitted .

Masters, J. C. and D. J. Brothers. 2002. Lack of congruence between morphological and molecular data in reconstructing the phylogeny of the galagonoidae. American Journal of Physical Anthropology 117:79–93.

Page, R. D. M. 2002. Modified mincut supertrees. Pages 537–552 *in* Second International Workshop on Algorithms in Bioinformatics (R. Guig and D. Gusfield, eds.) Springer.

Page, R. D. M. 2004. Taxonomy, supertrees, and the tree of life. Pages 247–265 *in* Phylogenetic supertrees: combining information to reveal the Tree of Life (O. R. P. Bininda-Emonds, ed.). Kluwer, Dordrecht.

Page, R. D. M. and G. Valiente. 2005. An edit script for taxonomy classifications. BMC Bioinformatics 6.

Roos, C., J. Schmitz, and H. Zischler. 2004. Primate jumping genes elucidates strepsirrhine phylogeny. Proceedings of the National Academy of Sciences 29:10650–10654.

Sanderson, M. J., B. G. Baldwin, G. Bharathan, C. S. Campbell, D. Ferguson, J. M. Porter, C. V. Dohlen, M. F. Wojciechowski, and M. J. Donoghue. 1993. The growth of phylogenetic information and the need for a phylogenetic database. Syst. Biol. 42:562–568.

Semple, C. and M. Steel. 2000. A supertree method for rooted trees. Discrete Appl. Math. 105:147–158.

Semple, C. and M. Steel. 2003. Phylogenetics. Oxford University Press, Oxford.

Yoder, A. D., R. M. Rasoloarison, S. M. Goodman, J. A. Irwin, S. Atsalis, M. J. Ravosa, and J. U. Ganzhorn. 2000. Remarkable species diversity in malagasy mouse lemurs (primates, microcebus). Proceedings of the National Academy of Sciences 97:11325–11330.

Zaroliagis, C. D. 2002. Implementations and experimental studies of dynamic graph algorithms. Pages 229–278 *in* Experimental algorithms: from algorithm design to robust and efficient software vol. 2547 of *Lect. Notes in Comput. Sci.* Springer.

Appendix 1: Correctness of AncestralBuild*

To ease reading in the following proofs, we view a node ℓ of a restricted descendancy graph that is not a most recent common ancestor label as a single-element set. Theorem 8 follows from Proposition 4 and Lemma 12.

Lemma 11 *Let $\mathcal{P}$ be a collection of rooted semi-labelled trees on X, and let $\mathcal{P}'$ be a collection of rooted fully-labelled trees that is obtained from $\mathcal{P}$ by adding most recent common*

ancestor labels. Let $\mathcal{T}_1 \in \mathcal{P}'$, and suppose that $a, b \in \mathcal{L}(\mathcal{T}_1) \cap X$ such that a and b are not comparable in $\mathcal{T}_1$. If $\ell \in \mathcal{L}(\mathcal{T}_1)$ is an ancestor label of both a and b in $\mathcal{T}_1$, then there is a path in $C(\mathcal{P}')$ from a to b in which all nodes on this path are descendant labels of ℓ in $\mathcal{T}_1$.

Proof. Without loss of generality, we may assume that ℓ labels the root of $\mathcal{T}_1$. Furthermore, since for any element $z \in \mathcal{L}(\mathcal{T}_1) \cap X$, there is a path in $C(\mathcal{P}')$ from z to any of its descendant labels in $\mathcal{L}(\mathcal{T}_1) \cap X$, we may also assume that $\mathcal{L}(\mathcal{T}_1) \cap X$ bijectively labels the leaves of $\mathcal{T}_1$. This implies that $\mathcal{T}_1$ has no degree-two nodes.

We prove the lemma by showing that, for any pair of elements x and y in $\mathcal{L}(\mathcal{T}_1) \cap X$, there is a path joining this pair in $C(\mathcal{P}')$ with the property that all nodes on this path are in $\mathcal{L}(\mathcal{T}_1) \cap X$. The proof is by induction on the distance d from $\mathrm{mrca}_{\mathcal{T}_1}(x, y)$ to the root of $\mathcal{T}_1$. Suppose that the height of $\mathcal{T}_1$ is h. If $d = h - 1$, then, by the definition of $C(\mathcal{P}')$, x and y are joined by an edge in $C(\mathcal{P}')$ and so the result holds. Now assume that $d = h - k$, where $2 \leq k \leq h$, and that, for all pairs of elements, w and z say, in $\mathcal{L}(\mathcal{T}_1) \cap X$ for which the distance from $\mathrm{mrca}_{\mathcal{T}_1}(w, z)$ to the root is greater than $h - k$, there is a path in $C(\mathcal{P}')$ from w to z that only uses elements in $\mathcal{L}(\mathcal{T}_1) \cap X$.

Let ℓ and ℓ' be the ancestor labels of x and y in $\mathcal{L}(\mathcal{T}_1)$ that label the sibling nodes of the node of $\mathcal{T}_1$ corresponding to the most recent common ancestor of x and y. Then, by the induction assumption, there is a path in $C(\mathcal{P}')$ from x to an element in ℓ using just elements in $\mathcal{L}(\mathcal{T}_1) \cap X$ and there is a path in $C(\mathcal{P}')$ from an element in ℓ' to y using just elements in $\mathcal{L}(\mathcal{T}_1) \cap X$. Furthermore, by the definition of the edge set of $C(\mathcal{P}')$, there is an edge joining each element in ℓ with each element in ℓ' in $C(\mathcal{P}')$. Hence there is a path from x to y in $C(\mathcal{P}')$ of the desired type. This completes the proof of the lemma. $\square$

Lemma 12 *Let $\mathcal{P}$ be a collection of rooted semi-labelled trees on X, and let $\mathcal{P}'$ be a collection of rooted fully-labelled trees that is obtained from $\mathcal{P}$ by adding most recent common ancestor labels. Suppose that* Descendant* *is applied to $D^*(\mathcal{P}')$ and $C(\mathcal{P}')$, and that at some iteration, i say, D_i and C_i are the inputted restrictions of $D^*(\mathcal{P}')$ and $C(\mathcal{P}')$, respectively, with $V(D_i) \cap X = V(C_i)$ and the following properties holding:*

- (i) *For all $a, c \in X$, there is a directed path of arcs in D_i from c to a with no element $x \in X$ as a non-terminal node in this path if and only if there is a type (i) edge joining c and a in C_i.*
- (ii) *The set of type (ii) edges of C_i is the disjoint union of sibling edge sets. Furthermore, if e is an edge of $D^*(\mathcal{P}')$ as a result of $\mathcal{T}_1 \in \mathcal{P}'$, then e is an edge in D_i if and only if each of the edges in the sibling edge set of e relative to $\mathcal{T}_1$ are in C_i.*

Let $\mathcal{S}_0$ denote the set of nodes of D_i with indegree zero and no incident edges. Then, in reference to Descendant*,

(I) *After Step 3′(a) is completed, if* $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k$ *are the node sets of the arc components of* $D_i \backslash \mathcal{S}_0$*, then* $\mathcal{S}_1 \cap X, \mathcal{S}_2 \cap X, \ldots, \mathcal{S}_k \cap X$ *are the node sets of the blue components of* $C_i \backslash (\mathcal{S}_0 \cap X)$*.*

(II) *Before Step 3′(c) is performed, an edge* $e = \{\ell, \ell'\}$ *of* $D_i \backslash \mathcal{S}_0$ *joins two arc components if and only if, for each sibling edge set of* e*, each edge in this set is coloured red and joins two blue components in* $C_i \backslash (\mathcal{S}_0 \cap X)$ *with the labels in* ℓ *in one blue component and the labels in* ℓ' *in the other blue component.*

(III) *After Step 3′(c) is completed, for each arc component of* $D_i \backslash \mathcal{S}_0$ *and the corresponding blue component of* $C_i \backslash (\mathcal{S}_0 \cap X)$*, (i) and (ii) still hold.*

Proof. We first prove (I). As every directed path in $D_i \backslash \mathcal{S}_0$ must end with an element of X, it follows that, for all i, we have that $\mathcal{S}_i \cap X$ is non-empty.

Let $\mathcal{U}$ be the node set of a blue component of $C_i \backslash (\mathcal{S}_0 \cap X)$. Let a and b be nodes in $\mathcal{U}$, and suppose that a and b are joined by a blue edge. Then either

(a) b is an ancestor of a or a is an ancestor of b in some tree in $\mathcal{P}$, or

(b) $\{a, b\}$ is an element of a sibling edge set of an edge, e say, in $D(\mathcal{P}')$.

Since D_i and C_i satisfy (i), it follows that in case (a), there is either a directed path from b to a or a directed path from a to b in $D_i \backslash \mathcal{S}_0$. In both cases, a and b are in the same arc component of $D_i \backslash \mathcal{S}_0$. If (b) holds, then, by (ii), e appears in $D_i \backslash \mathcal{S}_0$. Furthermore, as $\{a, b\}$ is blue, the parent node of $\{a, b\}$ in D_i is not an element of $\mathcal{S}_0$. It follows that, in $D_i \backslash \mathcal{S}_0$, there is an arc from this parent node to one end of e and an arc from this parent node to the other end of e. In particular, this means that a and b are in the same arc component of $D_i \backslash \mathcal{S}_0$. It now follows that $\mathcal{U}$ is a subset of the node set of some component, $\mathcal{S}_j$ say, of D_i.

To complete the proof, we next show that $\mathcal{S}_j \cap X$ is not the (disjoint) union of the node sets of two or more blue components of $C_i \backslash (\mathcal{S}_0 \cap X)$. To see this, assume that this happens. Then, in the arc component of $D_i \backslash \mathcal{S}_0$ whose node set is $\mathcal{S}_j$, there is a path of arcs (not necessarily a directed path) from a node c that is in the node set of one blue component of $C_i \backslash (\mathcal{S}_0 \cap X)$ to a node d that is in the node set of another blue component of $C_i \backslash (\mathcal{S}_0 \cap X)$. Without loss of generality, we may assume that amongst all such pairs of blue components in $C_i \backslash (\mathcal{S}_0 \cap X)$ and pairs of elements of X this path is of minimum length. By minimality, apart from c and d, there are no other elements of X on this path. But then each of the non-terminal nodes on this path are elements in $\mathcal{L}(\mathcal{P}') - X$. Since each of these labels across all trees in $\mathcal{P}'$ are distinct, we deduce that each of these labels come from the same tree, $\mathcal{T}_1$ say, in $\mathcal{P}'$. If d is an ancestor of c or c is an ancestor of d in $\mathcal{T}_1$, it follows by the minimality condition that $\{c, d\}$ is a blue edge in $C_i \backslash (\mathcal{S}_0 \cap X)$; a contradiction. So assume

that c and d are non-comparable in $\mathcal{T}_1$. By considering the directions of the arcs in the path of directed edges between c and d, it is easily seen that one node on this path, ℓ say, is an ancestor label of both c and d in $\mathcal{T}_1$. Since the node ℓ appears in $D_i\backslash\mathcal{S}_0$, it follows by the fact that $V(D_i) \cap X = V(C_i)$ that each of its descendant labels in $\mathcal{L}(T_1) \cap X$ are in $C_i\backslash(\mathcal{S}_0 \cap X)$. From Lemma 11, we now deduce that there is a path of blue edges in $C_i\backslash(\mathcal{S}_0 \cap X)$ from c to d, and so c and d are in the same blue component of $C_i\backslash(\mathcal{S}_0 \cap X)$. This contradiction completes the proof of (I).

To prove (II), first suppose that before Step 3′(c) is performed $e = \{\ell, \ell'\}$ is an edge of $D_i\backslash\mathcal{S}_0$ joining two arc components. Then no parent node of any sibling edge set of e is in the node set of $D_i\backslash\mathcal{S}_0$ and so, if it exists, every edge in each of these sets is coloured red in $C_i\backslash(\mathcal{S}_0 \cap X)$. Since $\{\ell, \ell'\}$ is an edge of $D_i\backslash\mathcal{S}_0$, every element in $\ell \cup \ell'$ is in the node set of $C_i\backslash(\mathcal{S}_0 \cap X)$ and so, by (ii), we deduce that all such edges exist. Now consider the elements in ℓ. If ℓ is a singleton, then, trivially, the labels in ℓ are in a single arc component. Therefore, assume that ℓ contains two labels x and y. Then ℓ corresponds to the most recent common ancestor of x and y in some tree in $\mathcal{P}'$. It follows that x and y are in the same arc component of $D_i\backslash\mathcal{S}_0$. Applying the same arguments to ℓ' and using (I), we deduce one direction of (II).

For the converse of (II), suppose that, for each sibling edge set of e, each of the edges in this set are coloured red and join two blue components in $C_i\backslash(\mathcal{S}_0 \cap X)$ with the labels in ℓ in one blue component and the labels in ℓ' in the other blue component. Then it is immediate from (I) that $\{\ell, \ell'\}$ joins two arc components in $D_i\backslash\mathcal{S}_0$. This completes the proof of (II).

For (III), consider a component D of $D_i\backslash\mathcal{S}_0$ and the corresponding component, C say, of $C_i\backslash(\mathcal{S}_0 \cap X)$. First assume that there is a directed path of arcs in D from c to a with no element $x \in X$ as a non-terminal node. Then, as $c \notin \mathcal{S}_0$, we have that c and a are elements of C. Since (i) holds for D_i and C_i, it immediately follows there is an arc joining c and a in C. A similar argument shows that the converse also holds. Thus D and C satisfy (i).

Now consider (ii) in the statement of the lemma. First observe that, as the type (ii) edges of C_i is the disjoint union of sibling edge sets, it follows by (II) and the way in which type (ii) edges of C_i and edges of D_i are deleted that the type (ii) edges of C is the disjoint union of sibling edge sets. Now assume that e is an edge in D with ends ℓ and ℓ'. Then, by the construction of $D^*(\mathcal{P}')$, it follows that $\ell \cup \ell'$ is a subset of the node set of D. As all of the elements in $\ell \cup \ell'$ are in X, we deduce by (I) that $\ell \cup \ell'$ is a subset of the node set of C. Since (ii) holds for D_i and C_i, it follows that each of the edges in the sibling edge sets of e are in C. A similar argument shows that the converse also holds. □

Appendix 2: Implementation Details

We give here a description of AncestralBuild* that is more tuned towards implementation. Data structures maintained throughout the algorithmic process are first detailed, then a pseudo-code details their precise interactions in the Descendant* subroutine.

Data structures

$C(\mathcal{P}')$ is a graph with node set X and whose edges can be one of two colours (blue or red). The nodes of $C(\mathcal{P}')$ are stored in an adjacency list. For each node v, we store its blue neighbours (*i.e.*, the nodes connected to v by a blue edge) in a doubly-linked list $B(v)$ and its red neighbors in a doubly-linked list $R(v)$. These neighbour relations code for the edges of $C(\mathcal{P}')$. Note that an edge $\{u, v\}$ of a given color, say blue, is represented twice, because both $u \in B(v)$ and $v \in B(u)$.

$D^*(\mathcal{P}')$ is a mixed graph with node set $\mathcal{L}(\mathcal{P}')$. For each node of $D^*(\mathcal{P}')$, we store its indegree, its list of out-going arcs, and the list of nodes to which it is connected by an edge. For each node, we also maintain a list of pointers to the sibling edges of which it is parent. For each sibling edge of $D^*(\mathcal{P}')$, we also maintain a list of pointers to the corresponding (one to four) edges in $C(\mathcal{P}')$. This pointer is doubled so that these edges in $C(\mathcal{P}')$ know in constant time their corresponding sibling edge in $D^*(\mathcal{P}')$. Thus, changing in $C(\mathcal{P}')$ the colour of sibling edges $\{u, v\}$ of a node $x \in D^*(\mathcal{P}')$ is done in constant time for each edge (recall that $B(v)$, and similarily $B(u)$, is a doubly-linked list, so that removing a pointed element out of it and putting it in $R(v)$ is done in constant time).

numC is a table indicating for each node s in $C(\mathcal{P}')$ the number of its current blue component in $C(\mathcal{P}')$. Initially, all elements belong to the same component (without loss of generality, we may assume that $C(\mathcal{P}')$ is initially connected).

Comp is a table storing the blue components of $C(\mathcal{P}')$. Each component is coded as a doubly-linked lists of nodes of X and is stored in the entry of corresponding to its number.

$\mathcal{S}$ is a table, whose i^{th} entry, $\mathcal{S}_i$, is the doubly-linked list of nodes of the ith arc component that have indegree 0 and no incident edge. If such a node of $D^*(\mathcal{P}')$ has a label in X, then there is a link between this node in $\mathcal{S}_i$ and the corresponding node of $C(\mathcal{P}')$ stored in the list $Comp_i$ which stores the nodes of the ith blue component. Thus, when a node of $C(\mathcal{P}')$ is moved from the ith blue component to the jth one (just created because of the deletion of an edge), the corresponding node of $D^*(\mathcal{P}')$ moves from $\mathcal{S}_i$ to $\mathcal{S}_j$ in constant time.

L_C^{next} is the list of new blue components in $C(\mathcal{P}')$ (corresponding to the new arc components in $D^*(\mathcal{P}')$) created during the on-going execution of Descendant*.

$\mathcal{A}$ is the dynamic connectivity algorithm supporting deletion updates and connectivity queries.

Pseudo-code for the Descendant* subroutine

The heading of the subroutine is the following:

Algorithm 1: Descendant* $(D^*(\mathcal{P}'), C(\mathcal{P}'), n_0)$

Input: The descendancy graph $D^*(\mathcal{P}')$, the component graph $C(\mathcal{P}')$, and an arc component of $D^*(\mathcal{P}')$ to process indicated by its number n_0.

Result: A tree with root labelled by $\mathcal{S}_{n_0} \cap X$ displaying a part of $\mathcal{P}'$, or the statement *no possible labelling*

1 **if** $\mathcal{S}_{n_0}$ *is empty* **then return** *no possible labelling*
2 **if** $\mathcal{C}_{n_0}$ *contains only one node* v **then**
    return the tree composed of one leaf with the label of v

First note that the whole graphs $D^*(\mathcal{P}')$ and $C(\mathcal{P}')$ are indicated as formal parameters, and not part of them. Indeed, to avoid unnecessary copies of parts of these graphs, it is simpler that all calls to the subroutine access the same shared data structure. Each call to the subroutine has to work on a single arc component of $D^*(\mathcal{P}')$ whose number n_0 is inputted to the subroutine.

3′(a)(i) $L_C^{next} \leftarrow \emptyset$; $\mathcal{S}_{n_0}^* \leftarrow \emptyset$;
foreach $s \in \mathcal{S}_{n_0}$ **do**
  delete s and every incident arc (s, x) in $D^*(\mathcal{P}')$;
  if *as a result, node* x *has indegree* 0 *(and no incident edge)* **then**
    $\mathcal{S}_{n_0}^* \leftarrow \mathcal{S}_{n_0}^* \cup \{x\}$ /* for next call on this component */

3′(a)(ii) **foreach** $s \in \mathcal{S}_{n_0} \cap X$ **do**
  remove s from $Comp_{numC(s)}$;
  foreach *edge* $\{s, x\} \in C(\mathcal{P}')$ **do**
    delete $\{s, x\}$ from $C(\mathcal{P}')$ and send to $\mathcal{A}$ the corresponding edge deletion update

3′(a)(iii) $L_{red} \leftarrow \emptyset$ /* list of edges coloured red in $C(\mathcal{P}')$ */;
foreach $s \in \mathcal{S}_{n_0}$ **do**
  let $Sib(s)$ be the set of sibling edges whose parent is s;
  foreach $e \in Sib(s)$ **do**
    let $L_{(ii)}(e)$ be the sibling edge set of $C(\mathcal{P}')$ corresponding to e;
    colour red all edges of $L_{(ii)}(e)$ in $C(\mathcal{P}')$;
    $L_{red} \leftarrow L_{red} \cup L_{(ii)}(e)$

$\mathcal{S}_{n_0} \leftarrow \mathcal{S}_{n_0}^*$;

Deleting nodes of $\mathcal{S}_{n_0}$ from $D^*(\mathcal{P}')$ can lead to other nodes of the n_0^{th} arc component to have indegree 0 (and no incident edge). However, these nodes are put aside (until the end of Step 3′(a)(iii)) in a list $\mathcal{S}_{n_0}^*$ and not directly in $\mathcal{S}_0$ to avoid confusion between the elements in that set at the beginning of the on-going call, and elements to be processed in the next recursive call for the same arc component.

Also recall that removing nodes from $C(\mathcal{P}')$ at Step 3′(a)(ii) (namely, nodes in $\mathcal{S}_{n_0} \cap X$) does not change the set of blue components of the graph.

```
3'(b) foreach edge {a, b} ∈ L_red do
        perform a deletion update in A for the edge {a, b};
        if A answers that a and b are in different components then
            C_a ← {a} and C_b ← {b};
            by examining blue edges in C(P') connected to nodes of C_a and C_b alterna-
            tively, determine other nodes in these blue components, halting the process
            when the smallest component, C_a say, is fully discovered;
            Let n_c be the number of the new component;
            foreach x ∈ C_a do
                remove x from Comp_numC(x) and add it to Comp_n_c;
                numC(x) ← n_c
            L_C^next ← L_C^next ∪ {C_a}
```

Step 3′(b) identifies new blue components of $C(\mathcal{P}')$ resulting from the red colouring given to some of its edges. When a former blue component is broken into two blue components, nodes of the smallest one are identified efficiently according to Even and Shiloach (1981). This smallest component receives a new number n_c and its nodes are transferred from the old component to the list in *Comp* corresponding to this new component.

```
3'(c) foreach C ∈ L_C^next do
        foreach node v of C do
            foreach red edge {v, v'} in C(P') do
                if numC(v) ≠ numC(v') then
                    remove {v, v'} from C(P') and from the list of its associated sibling
                    edge e in D*(P');
                    if {v, v'} was the last edge in the sibling edge set of e then
                        remove e from D*(P');
                        if a node x in D*(P') that e connected has now no more edges
                        incident to it and indegree 0 then
                            S_numC(x) ← S_numC(x) ∪ {x}
```

Because of Lemma 12(II), edges between arc components of $D^*(\mathcal{P}')$ correspond exactly to red edges between the corresponding blue components of $C(\mathcal{P}')$. At the beginning of the current call of DESCENDANT*, there is no red edge between two blue components in $C(\mathcal{P}')$. Thus, such red edges only exist between new blue components created at this step, more precisely, between those stored in L_C^{next}, and the other new (larger) blue components. This guides the code of Step 3′(c), see above.

4 **foreach** *component* C_i *in* $L_C^{next} \cup \{C_{n_0}\}$ **do**

    $res_i \leftarrow$ DESCENDANT* $(C(\mathcal{P}'), D^*(\mathcal{P}'), i)$;

    if res_i *returns no possible labelling* **then return** *no possible labelling*

return the tree obtained by grafting all res_i as child subtrees of a root node labelled by nodes in $\mathcal{S}_{n_0} \cap X$.

Note that, in the case where $\mathcal{P}'$ is compatible, Step 4 issues a recursive call for each new component created by the on-going execution of DESCENDANT*, as well as for the component C_{n_0}. Indeed, this latter component still contains nodes. Note that some of these nodes that were not in $\mathcal{S}_{n_0}$ at the beginning of the on-going call, can now be in this set because initial nodes of this set and some edges have been removed from this arc component.