Introduction to Computational Phylogenetics

Tandy Warnow
The University of Texas at Austin

No Institute Given

Table of Contents

1. Introduction

2. Trees
2.1 Rooted trees
2.2 Unrooted trees
2.3 Consensus trees
2.4 When trees are compatible
2.5 Measures of accuracy in estimated trees
2.6 Rogue taxa
2.7 Induced subtrees

3. Constructing trees from subtrees
3.1 Constructing trees from rooted triples
3.2 Constructing trees from quartet subtrees
3.3 General subtrees

4. Constructing trees from qualitative characters
4.1 Introduction
4.2 Constructing rooted trees from directed binary characters
4.3 Constructing unrooted trees from compatible binary characters
4.4 General issues in constructing trees from characters
4.5 Maximum compatibility
4.6 Maximum parsimony
4.7 Binary encoding of multi-state characters
4.8 Informative and uninformative characters

5. Constructing trees from distances
5.1 Step 1: computing distances
5.2 Step 2: computing a tree from a distance matrix

6. Statistical methods
6.1 Introduction to Markov models of evolution
6.2 Calculating the probability of a site pattern
6.3 Calculating the likelihood of a tree
6.4 Bayesian methods
6.5 Maximum likelihood methods

7. Other estimation issues

8. Reticulate evolution
8.1 Introduction
8.2 Phylogenetic networks in linguistics
8.3 Phylogenetic networks in biology

9. Reconciling gene trees

1 Introduction

This document includes the basic material needed to understand computational methods for estimating phylogenetic trees in biology and linguistics, and to read the literature critically.

Pre-requisites: We assume some background in algorithm design and analysis, and in proving algorithms correct. Thus, you should know how to calculate the running time of an algorithm, as well as the standard "big-oh" notation. You will also need to know what it means for a problem to be NP-hard or NP-complete, and for a problem to be polynomial time.

Much of the material involves probabilistic analysis of algorithms under stochastic models of evolution, so some very rudimentary probability theory is helpful.

Designing and studying programs for phylogeny and alignment estimation can be a very helpful complement to the theoretical component of the course, and programming assignments are suggested in various chapters. However, the material can be learned without doing any programming.

Finally, a critical reading of the scientific literature is an enormous aid in learning the field, and can be very revealing.

Topics from Discrete Mathematics that you will need to know include:

Hasse Diagrams and partially ordered sets. A partially ordered set (or "poset") is a set S with a binary relation $\leq$ defined on it, where $\leq$ is a partial order. Examples of partial orders include containment, where we write $X \leq Y$ if $X \subseteq Y$. Note that it is not the case that every two elements in the subset will be in relation (i.e., it is possible for neither $X \leq Y$ nor $Y \leq X$ to be true).

A Hasse Diagram is a directed graph that represents a partially ordered set. It is constructed by creating one node for each element in the poset, a directed edge from x to y (for $x \neq y$) if $x \leq y$, and then iteratively removing all directed edges $x \leq y$ if there is a vertex z such that $x \leq z$ and $z \leq y$.

Equivalence relations. A binary relation on a set S is a set of ordered pairs of elements in S. A binary relation R is said to be an equivalence relation if it satisfies the following properties:

- $< a, a >$ is in R for all elements a of S
- If $< a, b >$ is in R, then $< b, a >$ is in R
- If $< a, b >$ and $< b, c >$ are in R, then also $< a, c >$ is in R.

Graph theory. You will need to know graph terminology: nodes, vertices, edges, node degrees, connected, cycles, etc.

Organization of the textbook. The chapters are organized as follows. First, we define the new concepts, structures and algorithms, from a purely mathematical framework. After this, we show how these ideas are used in phylogenetic estimation - both in linguistics and in biology. References to more advanced literature are also provided, including references to scientific papers using these techniques.

2 Trees

Trees are graphs, with vertices (also known as "nodes") and edges. In the context in which we will consider them, they represent evolutionary histories, and may be called "phylogenies", "phylogenetic trees", or "evolutionary trees". Sometimes trees are drawn rooted, although (as we shall see) most methods for estimating evolutionary trees produce unrooted trees. This section is devoted to understanding the terminology regarding trees, learning how to move between rooted and unrooted versions of the same tree, how to determine whether two trees are the same or different, etc. This will turn out to be important in understanding how trees are constructed from **character** data.

2.1 Rooted Trees

We begin with a discussion of rooted trees. For a rooted tree T with leaf set S, we draw the tree with the root r on top, on the bottom, on the left, or on the right – implicitly giving the edges an orientation (usually away from the root, towards the leaves). In this document, we'll draw them as rooted at the top.

Even so, there are many ways of drawing trees, and in particular of representing the branching process within a tree. For example, in a rooted tree, a node may have more than two children, in which case it is called a "polytomy".

Definition 1. *A* **polytomy** *in a unrooted tree is a vertex with degree at least four. A vertex in an rooted tree with more than two children is also called a polytomy.*

Thus, we can also define a binary tree as one that does not contain any polytomies!

Definition 2. *A tree is said to be* **binary** *if it does not contain any polytomies. Thus, an unrooted tree is said to be a* **binary tree** *if it does not contain any nodes of degree four or larger, and a rooted tree is said to be a binary tree if it does not contain any nodes with more than two children.*

The representation of a polytomy can vary between different graphical representations. In Figure 1, we show two equivalent representations of the same branching process. One of these (on the left) is standard in computer science, and the other (on the right) is often found within biological systematics. Note that in Figure 1(b), the horizontal lines do not necessarily correspond to edges.

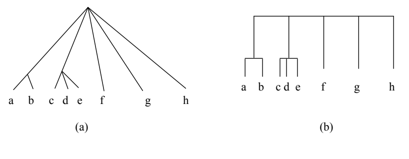


Fig. 1. Two ways of drawing the same tree

Graphical representations of trees sometimes include branch lengths, to help suggest relative rates of change and/or actual amounts of elapsed time. The "topology" of the tree is independent of the branch lengths, however, and is generally speaking the primary interest of the systematist.

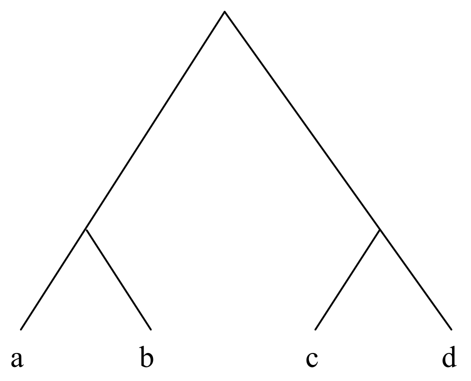


Fig. 2. Tree ((a,b),(c,d))

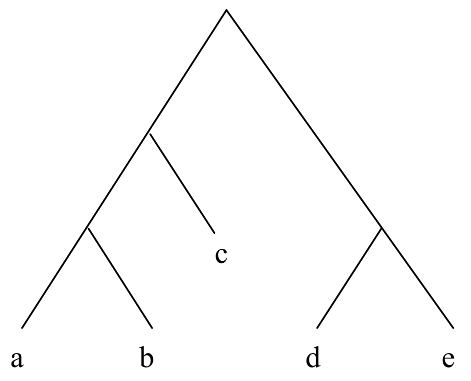


Fig. 3. Tree (((a,b),c),(d,e))

Newick Notation for rooted trees The first task is to be able to represent trees using Newick format: $((a, b), (c, d))$ represents the rooted tree with four leaves, a, b, c, d, with a and b siblings on the left side of the root, and c and d siblings on the right side of the root.

The same tree could have been written $((c, d), (a, b))$, or $((b, a), (d, c))$, etc. Thus, the graphical representation is somewhat flexible – swapping sibling nodes (whether leaves or internal vertices in the tree) doesn't change the tree "topology". As a result, there are *only three* rooted trees on leaf-set $\{a, b, c\}$!

Similarly, the following Newick strings refer to exactly the same tree as in Figure 3: $((d, e), (c, (a, b)))$, $((e, d), ((a, b), c))$, $((e, d), (c, (a, b)))$. Similarly, there are exactly 8 different Newick representations for the tree given in Figure 2:

– $((a, b), (c, d))$
– $((b, a), (c, d))$
– $((a, b), (d, c))$
– $((b, a), (d, c))$
– $((c, d), (a, b))$
– $((c, d), (b, a))$
– $((d, c), (a, b))$
– $((d, c), (b, a))$

The second fundamental task is to be able to recognize when two rooted trees are the same. Thus, when you don't consider branch lengths, the trees given in Figures 3 through 5 are different drawings of the same basic tree.

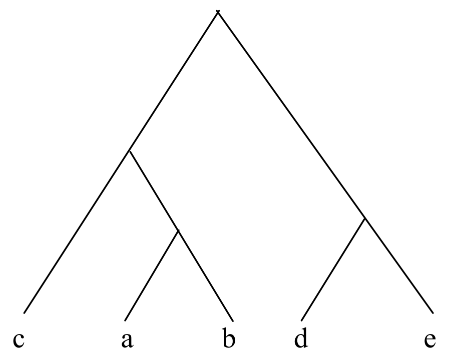


Fig. 4. Another drawing of the tree in Figure 3

The clade representation of a rooted tree. One way to determine if two rooted trees are the same or different is to write down their *clades*, where a clade in a tree is a maximal set of leaves that all have the same most recent common ancestor.

Definition 3. *Let T be a rooted tree in which every leaf is labelled by a distinct element from a set S. Let v be a node in T, and let X_v be the leaf set in the subtree of T rooted at v. Then X_v is a* **clade** *of T.*

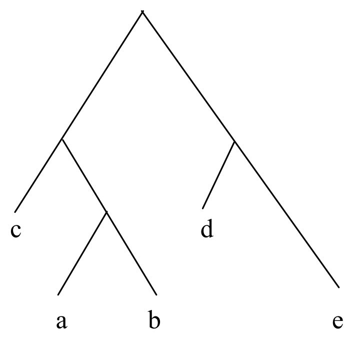


Fig. 5. Yet another drawing of the tree in Figure 3

To generate all the clades in a rooted tree, look at each node in turn, and write down the leaves below that node.

Definition 4. *Let T be a rooted tree on leaf-set S, and let X_v denote the set of leaves in the subtree of T rooted at vertex v. We define the set $Clades(T) = \{X_v : v \in V(T)\}$. Thus, $Clades(T)$ has all the singleton sets (each containing one leaf), a set containing all the taxa (defined by the root of T), and a set for each internal node other than the root that contains a subset of S with at least 2 but less than $n = |S|$ taxa.*

Thus, for the tree $T = ((a, b), (c, (d, e))$, the set $Clades(T)$ is given by

$$\text{Clades(T)} = \{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{a, b\}, \{d, e\}, \{c, d, e\}, \{a, b, c, d, e\}\}.$$

To determine if two trees T and T' are the same, you can write down the set of clades for the two trees, and see if the sets are identical. If $Clades(T) = Clades(T')$, then $T = T'$; otherwise, $T \neq T'$.

The set of clades of a tree always includes the singleton sets (consisting of $\{x\}$, for each taxon x), as well as the set of all the taxa. These clades are called the "trivial" clades, since they appear in every tree, and the other clades are called "non-trivial clades".

Constructing a rooted tree from its set of clades We now show how to compute a tree from its set of clades. To do this, we draw the "Hasse Diagram" for the "poset" (partially ordered set) defined by the clades, ordered by inclusion. That is, make a graph, with all the clades (including the trivial clades, including the full set of taxa) as nodes in the graph. Draw a directed edge from a node x to a different node y if $x \subset y$. Since containment is transitive, if $x \subset y$ and $y \subset z$, then $x \subset z$. Hence, if we have directed edges from x to y, and from y to z, then we know that $x \subset z$, and so can remove the directed edge from x to z without loss of information. This is the basis of the Hasse Diagram: you take the graphical representation of a partially ordered set, and you remove directed edges that are implied by transitivity. Furthermore, if we begin with the set of clades of a tree, then the resultant graph is the tree itself. Thus, constructing the Hasse Diagram for a set of clades of a tree produces the tree itself (provided that we include all the clades, not just the non-trivial clades).

You can run the algorithm on an arbitrary set of subsets of a taxon set S, but the output may or may not be a tree. For example, consider the following input:

– $A = \{\{a\}, \{a,b,c,d\}, \{a,d,e,f\}, \{a,b,c,d,e,f\}\}$.

On this input, there are four sets, and so the Hasse Diagram will have four vertices. Let v_1 denote the set $\{a\}$, v_2 denote the set $\{a,b,c,d\}$, v_3 denote the set $\{a,d,e,f\}$, and v_4 denote the set $\{a,b,c,d,e,f\}$. Then, in the Hasse Diagram, we will have the following directed edges: $v_1 \to v_2$, $v_1 \to v_3$, $v_2 \to v_4$, and $v_3 \to v_4$. This is not a tree, since it has a cycle (even though this is only a cycle when considering the graph as an undirected graph).

Theorem 1. *Let T be a rooted tree in which every internal node has at least two children. Then the Hasse Diagram constructed for $Clades(T)$ is identical to T.*

Proof. We prove this by induction on the number n of leaves in T. For $n = 1$, then T consists of a single node (since every node has at least two children). When we construct the Hasse Diagram for T, we obtain a single node, which is the same as T.

The Inductive Hypothesis is that the statement is true for all positive n up to $N-1$. We now consider a tree T with N leaves. Since the root of T has at least two children, we denote the subtrees of the root as $t_1, t_2, \ldots, t_k$ (with $k \geq 2$). Note that $Clades(T) = \bigcup_i Clades(t_i) \cup \{Leaves(T)\}$. Note also that the vertices for the Hasse Diagram on T come from vertices for the Hasse Diagrams on each t_i, and one node for $\{Leaves(T)\}$. Every directed edge in the Hasse Diagram on T is either a directed edge in the Hasse Diagram on some t_i, or is the directed edge from $\{Leaves(t_i)\}$ to $\{Leaves(T)\}$. By the inductive hypothesis, the Hasse Diagram defined on $Clades(t_i)$ is identical to t_i. Hence the Hasse Diagram defined on $Clades(T)$ is identical to T.

Compatible sets of clades: In the previous material, we have assumed we were given the set $Clades(T)$, and we wanted to construct the tree T from that set. Here we consider a related question: given a set X of subsets of a set S of taxa, is there a tree T so that $X \subseteq Clades(T)$? To answer this, apply the algorithm above, and see if you get a tree! For example, if we try this on $\{\{a,b\}, \{b,e\}, \{c,d\}\}$, we do not get a tree. Therefore, this set of subsets is *not* the set of clades of any tree. When the set of subsets *is* a subset of the set of clades of a tree, we say that the set of subsets is *compatible*, and otherwise we say it is *not compatible*.

Definition 5. *A set X of subsets is said to be* **compatible** *iff there is a rooted tree T with each leaf in T given a different label, so that $X \subseteq Clades(T)$.*

Lemma 1. *A set A of subsets is compatible if and only if for any two elements X and Y in A, either X and Y are disjoint or one contains the other.*

Proof. If a set A of subsets is compatible, then there is a rooted tree T on leaf set S, in which every leaf has a different label, so that each element in A is the set of leaves below some vertex in T. Let X and Y be two elements in A, and let x be the vertex of T associated to X and y be the vertex associated to Y. If x is an ancestor of y, then X contains Y, and similarly if y is an ancestor of x then Y contains X. Otherwise neither is an ancestor of the other, and the two sets are disjoint.

For the reverse direction, note that when all pairs of elements in set A satisfies this property, then the Hasse Diagram will be a tree T so that $A = Clades(T)$.

The following corollary follows immediately, and will be very useful in algorithm design!

Corollary 1. *A set A of subsets of S is compatible if and only if every pair of elements in A are compatible.*

Difficulties in rooting trees Although evolutionary trees are rooted, estimations of evolutionary trees are almost always unrooted, for a variety of reasons. In particular, unless the taxa (languages, genes, species, whatever) evolve under a "strong clock" (so that the expected number of changes is proportional to the time elapsed since a common ancestor), rooting trees requires additional information. The typical technique is to use an "outgroup" (a taxon which is not as closely related to the remaining taxa as they are to each other). The outgroup taxon is added to the set of taxa and an unrooted tree is estimated on the enlarged set. This unrooted tree is then rooted by "picking up" the unrooted tree at the outgroup. See Figure 6, where we added a fly to a group of mammals. If you root the tree at the fly, you obtain the rooted tree $(cow, (chimp, human))$, showing that chimp and human have a more recent common ancestor than cow has to either human or chimp.

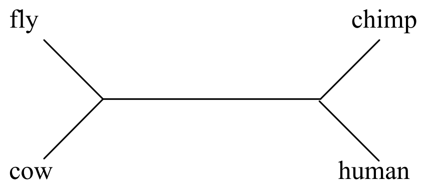


Fig. 6. Tree on some mammals with fly as the outgroup

The problem with this technique is subtle: while it is generally easy to pick outgroups, the less closely related they are to the remaining taxa, the less accurately they are placed in the tree. That is, very distantly related taxa tend to fit equally well into many places in the tree, and thus produce incorrect rootings. See Figure 7, where the outgroup (marked by "outgroup") attaches into two different places within the tree on the remaining taxa. Note how the trees on the remaining taxa are different as rooted trees (when rooted at the outgroup), although identical as unrooted trees.

Furthermore, it is often difficult to distinguish between an outgroup taxon that is closely related to the ingroup taxa, and a taxon that is, in fact, a member of the same group which branched off early in the group's history. For this reason, even the use of outgroups is somewhat difficult.

2.2 Unrooted trees

We now turn to discussions of unrooted trees. We begin with writing down rooted versions of unrooted trees, and then writing down unrooted versions of rooted trees.

Newick formats for unrooted trees First, the Newick format that is used to represent a rooted tree is also used to represent its unrooted version. In other words, every unrooted tree will have several Newick representations, for each of the ways of rooting the unrooted tree. Since phylogeny estimation

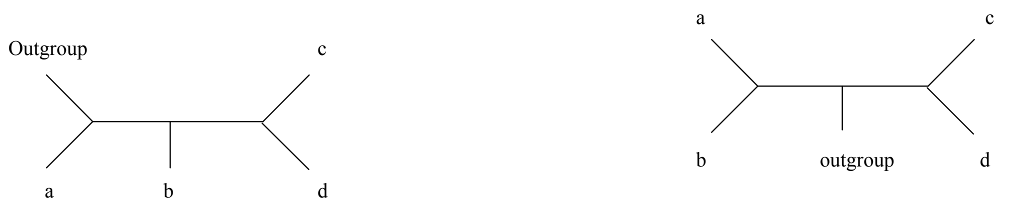


Fig. 7. Two trees which differ only in the placement of the outgroup

methods almost universally produce unrooted trees, although the output of a phylogeny estimation procedure may be given in a rooted form, the particular location of the root is irrelevant and should be ignored.

Now that you know how to draw unrooted versions of rooted trees, we will do the reverse. You can generate rooted trees from an unrooted tree by picking up the tree at any edge, or at any node. You can even pick up the tree at one of its leaves, but then the tree is rooted at one of its own taxa – which we generally don't do (in that case, we'd root it at the edge leading to that leaf instead, thus keeping the leaf set the same). Suppose we consider the unrooted tree given in Figure 8, which has four leaves: a, b, c, d, where a and b are siblings, and c and d are siblings.

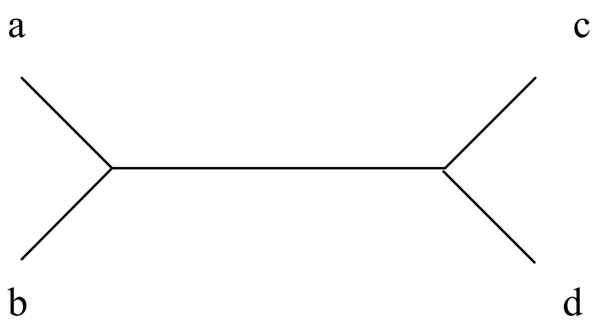


Fig. 8. Drawing of the unrooted tree ((a,b), (c,d))

This tree has 5 edges and two internal nodes. If we root the tree at one of the internal nodes, we will get a rooted tree with three children, while rooting the tree at an edge gives a rooted tree in which all nodes have two children.

More generally, if we root a binary unrooted tree (i.e., an unrooted tree in which all internal nodes have degree three) on an edge, we obtain a rooted binary tree.

Definition 6. *Every node in a tree is either a* **leaf** *(in which case it has degree one) or an* **internal node***.*

Two of the rooted trees consistent with the unrooted tree given in Figure 8 are provided in Figures 9 and 10.

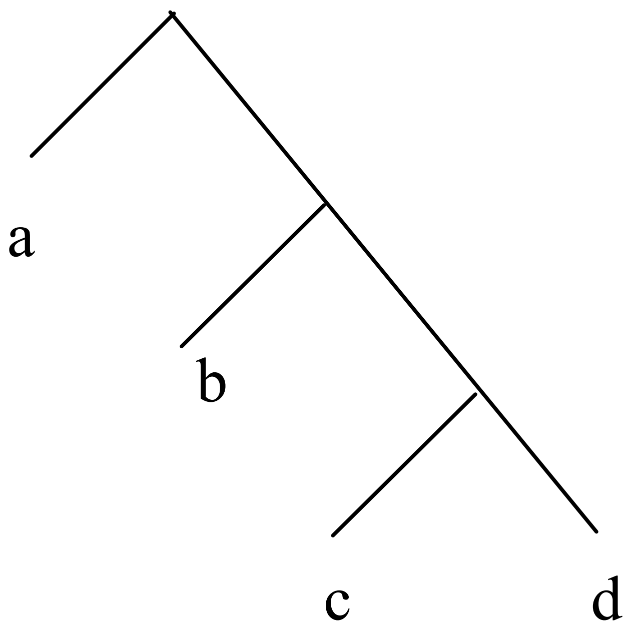


Fig. 9. One rooted version of ((a,b), (c,d)).

The bipartitions of an unrooted tree To determine if two unrooted trees are the same, we do something similar to what we did to determine if two rooted trees are the same. However, since the trees are unrooted, we cannot write down clades. Instead, we write down "bipartitions".

The bipartitions of an unrooted tree are formed by taking each edge in turn, and writing down the two sets of leaves that would be formed by deleting that edge. Note that when the edge is incident to a leaf, then the bipartition is trivial – it splits the set of leaves into one set with a single leaf, and the

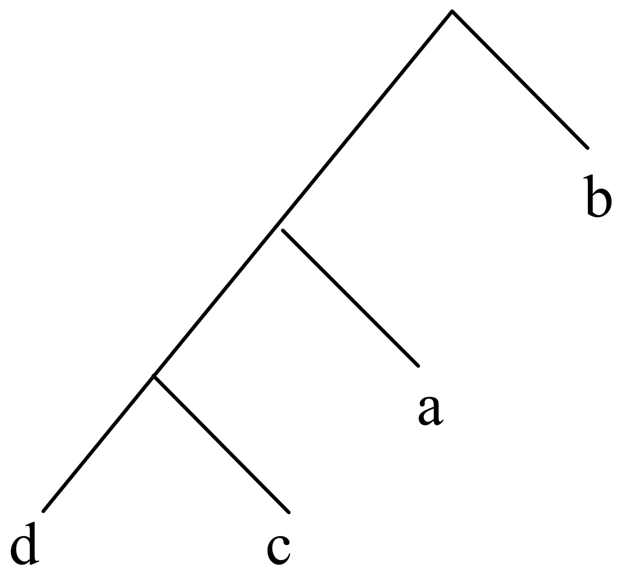


Fig. 10. Another rooted version of ((a,b), (c,d)).

other set with the remaining leaves. These bipartitions are present in all trees with any given leaf set. Hence, we will focus just on the non-trivial bipartitions.

For the tree in the previous section with four leaves a, b, c and d, there was only one non-trivial bipartition, splitting a and b on one side from c and d on the other. We denote this bipartition by $\{\{a,b\}|\{c,d\}\}$, or more simply by $(ab|cd)$. Note that we could have denoted this by $(cd|ab)$ or $(dc|ab)$, etc; the order in which the taxa appear within any one side does not matter, and you can put either side first. Note also that we can omit commas, as long as the meaning is clear. We will call the set of non-trivial bipartitions derived in this way the "bipartition encoding" of the tree, and denote it by $C(T)$. (We will also refer to this as the "character encoding"; see later.)

We summarize this discussion with the following definition:

Definition 7. *Given an unrooted tree T with no nodes of degree two, the* **bipartition encoding** *of T, denoted by $C(T) = \{\pi(e) : e \in E(T)\}$, is the set of bipartitions defined by each edge in T, where $\pi(e)$ is the bipartition on the leaf set of T produced by removing the edge e (but not its endpoints) from T.*

Comparing trees using their bipartitions It is easy to see that we can write down the set of bipartitions of any given unrooted tree, and that two unrooted trees are identical if they have the same set of bipartitions. However, other relationships can also be inferred: for example, we can see when one tree *refines* another, by comparing their bipartitions. That is, if T and T' are two trees on the same leaf set, then T is said to *refine* T' if T' can be obtained from T by contracting some edges in T. In fact, T refines T' if and only if $C(T') \subseteq C(T)$. (Note that using this definition, each tree refines itself, and is also a contraction of itself, since we can choose to contract no edges.)

Definition 8. *Given two trees T and T' on the same set of leaves (and each leaf given a different label), tree T is said to* **refine** *T' if T' can be obtained from T by contracting a set of edges in T. We also express this by saying T* **is a refinement of** *T' and T'* **is a contraction of** *T.*

Constructing T from $C(T)$: Sometimes we are given a set A of bipartitions, and we are asked whether these bipartitions could co-exist within a tree (i.e., whether there exists a tree T so that $A \subseteq C(T)$). When this is true, the set of bipartitions is said to be *compatible*, and otherwise the set is said to be *incompatible.*

Definition 9. *A set A of bipartitions is* **compatible** *if there exists a tree T in which every leaf has a distinct label from a set S, so that $A \subseteq C(T)$.*

As we will see, if for a given set A is compatible, then in fact there is a tree T so that $A = C(T)$, and we can construct that tree T (which will be unique, assuming there are no nodes of degree two) in polynomial time. First, pick any leaf (call it "r") in the set to function as a root. This has the result of turning the unrooted tree into a rooted tree, and *therefore* turns the bipartitions into *clades.* For each bipartition, we write down the subset which does not contain r, and denote it as a clade. The set of clades that is produced in this fashion is then used to construct the rooted tree, using the technique given above.

Note that the tree we compute in this way does not include r (and will *also* not include any leaf that appears on the same side as r in every bipartition). Therefore, at the end, we add the leaf r and all the other leaves that always appear with r, as the root to the entire tree (separated from the rest of the tree by an edge!), and then we *redraw it* as an unrooted tree.

Equivalently, before you construct the Hasse Diagram, you can add all the trivial clades to the set of clades you have obtained. This would mean you would add one clade for each taxon, including the root r and the nodes that always appear with r, and also the clade that contains all the taxa. Then you can construct the Hasse Diagram for this enlarged set of clades.

Both techniques will produce a rooted tree T that you can then unroot to obtain your solution.

Example of this technique: We provide an example of this technique on the following input:

$$A = \{(123|456789), (12345|6789), (12|3456789), (89|1234567)\}.$$

First, we decide to root the tree at leaf 1. We note that 1 and 2 are always on the same side of every bipartition, and so we are actually rooting the tree at both 1 and 2. The set of clades we derive is

$$Clades(T) = \{\{4,5,6,7,8,9\}, \{6,7,8,9\}, \{3,4,5,6,7,8,9\}, \{8,9\}\}.$$

The Hasse Diagram we obtain for this set is given by (3,(4,5,(6,7,(8,9)))), and has the graphical representation given in Figure 11. If we then add leaves 1 and 2 as the root, we obtain the rooted tree

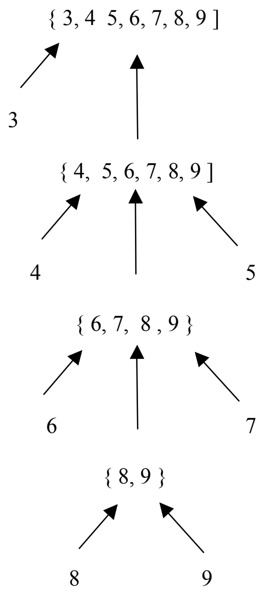


Fig. 11. Hasse Diagram.

given in Figure 12. We then unroot this tree, to obtain the tree given in Figure 13.

Now that we have constructed this unrooted tree, we check that it has the requisite bipartitions.

Pairwise Compatibility ensures Setwise Compatibility: Just as we saw with testing compatibility for clades, it turns out that bipartition compatibility has a simple characterization, and pairwise compatibility ensures setwise compatibility.

Theorem 2. *A set A of bipartitions on a set S is compatible if and only if every pair of bipartitions is compatible. Furthermore, a pair $X = (X_1, X_2)$ and $Y = (Y_1, Y_2)$ of bipartitions is compatible if and only if at least one of the four pairwise intersections $X_i \cap Y_j$ is empty.*

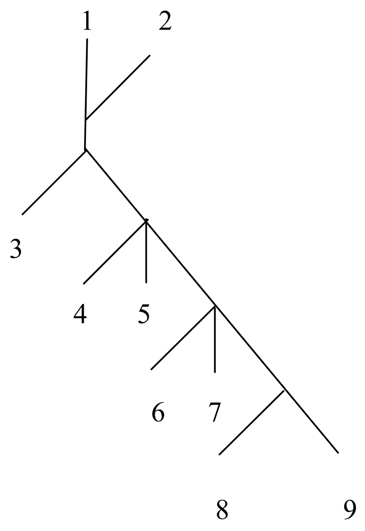


Fig. 12. Rooted tree on taxa 1, 2, ..., 9. Note the edge separating the root set {1,2} from the rest of the tree.

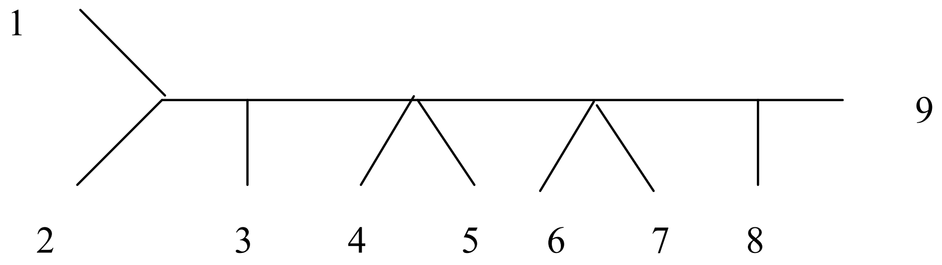


Fig. 13. Unrooted tree on 1...9, obtained by unrooting the tree in Figure 12.

Proof. We begin by proving that a pair of bipartitions is compatible if and only if at least one of the four pairwise intersections is empty. It is easy to see that a pair of bipartitions is compatible if and only if the clades produced (for any way of selecting the root) are compatible. So let's assume that we set s to be the root (for an arbitrary element $s \in S$, and that $s \in X_1 \cap Y_1$. Therefore, X and Y are compatible as bipartitions if and only if X_2 and Y_2 are compatible as clades. Therefore, X and Y are compatible as bipartitions if and only if one of the following statements holds:

- $X_2 \subseteq Y_2$
- $Y_2 \subseteq X_2$
- $X_2 \cap Y_2 = \emptyset$

If the first condition holds, then $X_2 \cap Y_1 = \emptyset$, and at least one of the four pairwise intersections is empty. Similarly, if the second condition holds, then $Y_2 \cap X_1 = \emptyset$, and at least one of the four pairwise intersections is empty. If the third condition holds, then directly at least one of the four pairwise intersections is empty. Thus, if X and Y are compatible as bipartitions, then at least one of the four pairwise intersections is empty.

For the converse, suppose that X and Y are bipartitions on S, and at least one of the four pairwise intersections is empty; we will show that X and Y are compatible as bipartitions. Let s be any taxon in S, and assume that $s \in X_1 \cap Y_1$. Hence, to show that X and Y are compatible as bipartitions it will suffice to show that X_2 and Y_2 are compatible as clades. Since $X_1 \cap Y_1 \neq \emptyset$, the pair that produced the empty intersection must be one of the other pairs: i.e., one of the following must be true: $X_1 \cap Y_2 = \emptyset, X_2 \cap Y_2 = \emptyset$, or $X_2 \cap Y_1 = \emptyset$. If $X_1 \cap Y_2 = \emptyset$, then $Y_2 \subset X_2$, and X_2 and Y_2 are compatible clades; thus, X and Y are compatible bipartitions. If $X_2 \cap Y_1 = \emptyset$, then a similar analysis shows that $Y_2 \subseteq X_2$, and so X and Y are compatible bipartitions. Finally, if $X_2 \cap Y_2 = \emptyset$, then directly X_2 and Y_2 are compatible clades, and so X and Y are compatible bipartitions.

Now that we have established that two bipartitions are compatible if and only if at least one of the four pairwise intersections is empty, we show that a set of bipartitions is compatible if and only if every pair of bipartitions is compatible. So let $s \in S$ be selected arbitrarily as the root, and consider all the clades (halves of bipartitions) that do not contain s. This set of subsets of S is compatible if and only if every pair of subsets is compatible, by Theorem 1. Hence, the theorem is proven.

2.3 Consensus trees

When two or more trees are given on the same leaf set, we may also be interested in computing *consensus trees*. In general, these consensus methods are applied to unrooted trees (and we will define them in that context), but they can be modified so as to be applicable to rooted trees as well.

Although there are many consensus tree methods, we will focus on the ones that are the most frequently used in practice:

- Strict Consensus
- Majority Consensus
- Greedy Consensus

To construct the strict consensus, we write down the bipartitions that appear in *every* tree in the input (the "profile"). The tree which has exactly that set of bipartitions is the "strict consensus". Note that the strict consensus is a contraction of *every tree* in the input (though if all the trees are identical, then it will be equal to them all).

Definition 10. *Given a set $\{T_2, T_2, \ldots, T_k\}$ of unrooted trees, each on the same leafset, the* **strict consensus** *tree T is the tree that contains exactly the bipartitions that appear in all the trees. Therefore, $C(T) = \cap_i C(T_i)$.*

To construct the majority consensus, we write down the bipartitions that appear in *more than half* the trees in the profile. The tree that has exactly those bipartitions is called the "majority consensus" (note that we mean strict majority).

Definition 11. *Given a set $\{T_2, T_2, \ldots, T_k\}$ of unrooted trees, each on the same leafset, the* **majority consensus** *tree T is the tree that contains exactly the bipartitions that appear in more than half of the trees.*

Observation 3 *The majority consensus is either equal to the strict consensus, or it refines the strict consensus, since it has every bipartition that appears in the strict consensus.*

We now define the greedy consensus, by showing how to compute it. To construct the greedy consensus, we order the bipartitions by the frequency with which they appear in the profile. We then start with the majority consensus, and then "add" bipartitions, one by one, to the tree we've computed so far. We stop either when we construct a fully resolved tree (because in that case no additional bipartitions can be added), or because we finish examining the entire list.

Note that the order in which we list the bipartitions will determine the greedy consensus – so that this particular consensus is *not uniquely defined* for a given profile of trees. On the other hand, the strict consensus and majority consensus do *not* depend upon the ordering, and are uniquely defined by the profile of trees.

Observation 4 *The greedy consensus is either equal to the majority consensus or it refines it, since it has every bipartition that appears in the majority consensus. Therefore, the greedy consensus is also called the* **extended majority consensus**.

Example: We give three different trees on the same leaf set:

- T_1 given by $C(T_1) = \{(12|3456), (123|456), (1234|56)\}$
- T_2 given by $C(T_2) = \{(12|3456), (123|456), (1235|46)\}$
- T_3 given by $C(T_3) = \{(12|3456), (126|345), (1236|45)\}$

The bipartitions are:

- (12|3456), which appears three times
- (123|456), which appears twice
- (1234|56), which appears once
- (1235|46), which appears once
- (1236|45), which appears once
- (126|345), which appears once

Using the definition of the strict consensus tree, we see that the strict consensus has only one bipartition, (12|3456). On the other hand, the majority consensus has two bipartitions – (123|456) and (12|3456). Finally, we discuss the greedy consensus. Note that this consensus tree depends upon the ordering of the remaining four bipartitions (since all appear exactly once, they can be ordered arbitrarily), and so there can be more than one greedy consensus tree. In fact, there are 24=4! possible ordering of these bipartitions! However, we will only show the results for three of these.

- Ordering 1: $(1234|56), (1235|46), (1236|45), (126|345)$. For this ordering, we see that we can add (1234|56) to the set we have so far, $(12|3456), (123|456)$, to obtain a fully resolved tree. Note that this is equal to T_1.

– Ordering 2: $(126|345), (1236|45), (1234|56), (1235|46)$. For this ordering we see that we *cannot* add the bipartition $(126|345)$ to the set we have so far. However, we can add $(1236|45)$, to obtain a fully resolved tree. This final tree is given by $(1,(2,(3,(6,(4,5)))))$. This is not among the trees in the input.
– Ordering 3: $(126|345), (1235|46), (1234|56), (1236|45)$. For this ordering, we cannot add $(126|345)$, but we can add the next bipartition, $(1235|46)$. When we add this, we obtain a fully resolved tree which is equal to T_2.

2.4 When trees are compatible.

Finally, we may be interested in combining the input trees into a single tree on the entire set of taxa, without using consensus methods. For example, when the set of trees has a common refinement, we would like to find that common refinement. In this case, we say that the trees are *compatible*, and we call the common refinement tree the *compatibility tree.*

Definition 12. *Let $\{T_1, T_2, \ldots, T_k\}$ be a set of unrooted trees all on the same set of leaves. If there exists a tree T that is a common refinement of all the T_i, then the minimally resolved tree T that is a common refinement is said to be the* **compatibility tree***.*

As an example, the following trees are compatible:

– T_1 given by $C(T_1) = \{(abc|defg)\}$, and shown in Figure 14.
– T_2 given by $C(T_2) = \{(abcd|efg), (abcde|fg)\}$, and shown in Figure 15.

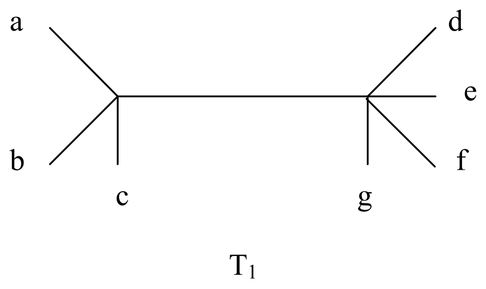


Fig. 14. Tree T_1

We can see they are compatible, because the tree in Figure 16 is a common refinement of each of the trees. However, there are other common refinements of these two trees; for example, see the tree in Figure 17.

Now consider the strict consensus of all the common refinements of trees T_1 and T_2. What does it look like? What bipartitions must it have? This *minimal* common refinement of these two trees is called the *compatibility tree*, and its character encoding is identical to the union of the character encodings of

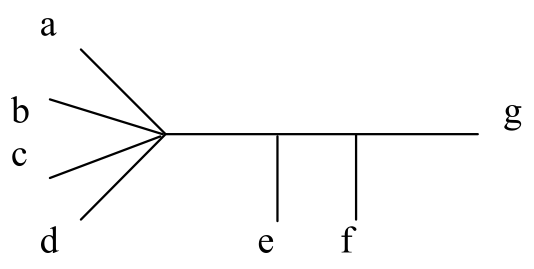


Fig. 15. Tree T_2

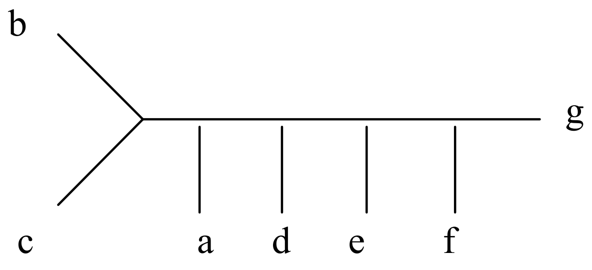


Fig. 16. Tree that is compatible with T_1 and T_2

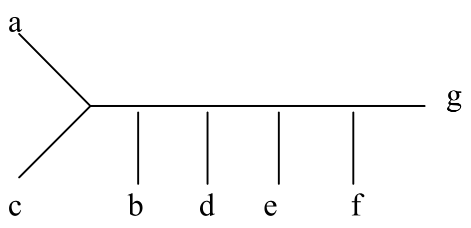


Fig. 17. Another common refinement of T_1 and T_2

the two trees! Thus, we can construct that minimal common refinement by computing the tree whose character encoding is that union, using the algorithm given in the previous sections.

Lemma 2. *If a set of trees $\{T_1, T_2, \ldots, T_k\}$ has a compatibility tree T, then $C(T) = \bigcup_i C(T_i)$.*

More generally, to see if a *set* of trees is compatible, we write down their bipartition sets, and then we apply the algorithm for constructing trees from bipartitions to the *union* of these sets. This will produce the compatibility tree, if it exists. If it fails to construct a tree, it proves that the set is not compatible.

Theorem: A set $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ of trees on the same leaf set is compatible if and only if the set $\cup_i C(T_i)$ is compatible.

Linguistic examples In linguistic analyses, different sets of characters may yield different trees, but if all the analyses are correct, then the differences *should* be only in terms of which edges are resolved, and which are not. That is, the resultant trees *should* be compatible.

2.5 Measures of accuracy in estimated trees

The context in which we will be interested in trees is where we are estimating trees from data, but are hoping to come "close" to the true tree. Since the true tree is unknown, determining how close we have come is often difficult. However, for the purposes of this section, we will presume that the true tree is known, so that we can compare estimated trees to the true tree.

Let us presume that the tree T_0 on leaf set S is the true tree, and that another tree T is an estimated tree for the same leaf set. There are several techniques that have been used to quantify errors in T with respect to T_0, of which the dominant ones are these:

False Negatives (FN): The false negatives are those edges in T_0 inducing bipartitions that do not appear in $C(T)$; this is also called the "missing branch" or "missing edge" rate. The false negative rate is the fraction of the total number of non-trivial bipartitions that are missing, or $\frac{|C(T_0)-C(T)|}{|C(T_0)|}$.

False Positives (FP): The false positives in a tree T with respect to the tree T_0 are those edges in T that induce bipartitions that do not appear in $C(T_0)$. The false positive rate is the fraction of the total number of non-trivial bipartitions in T that are false positives, or $\frac{|C(T)-C(T_0)|}{|C(T)|}$.

Robinson-Foulds (RF): The most typically used error metric is the sum of the number of false positives and false negatives, and is called the Robinson-Foulds distance. This distance ranges from 0 (so the trees are identical) to at most $2n-6$, where n is the number of leaves in each tree. To turn this into an error rate, that number is divided by $2n-6$ (see below for a discussion about this).

Comments: A few comments are worth making here. First, most typically, evolutionary trees are presumed to be *binary*, so that all internal nodes have three neighbors (or, if rooted, then every internal node has two children). In this case, the number of internal edges in the tree is $n-3$, and false negative error rates are produced by dividing by $n-3$. When both the estimated and true trees are binary, then false negative and false positive rates are equal, and these also equal the Robinson-Foulds distance. The main advantage in splitting the error rate into two parts (false negative and false positive) is that many estimated trees are not binary. In this case, when the true tree is presumed to be binary, the false positive error rate will be less than the false negative error rate. Note also that the reverse can happen – the false negative error rate could be smaller than the false positive error rate – when the true tree is not binary. Also note that because Robinson-Foulds distances are normalized by dividing by $2n-6$, they are *not* equal to the average of the false negative and false positive error rates. Also, the RF rate of a star tree (one with no internal nodes) is 50%, which is the same as the RF rate for a completely resolved tree that has half of its edges correct. Using the RF rate has been criticized because of this phenomenon, since it tends to favor unresolved trees.

Finally, the following can be established:

Observation 5 *Let T be the true tree, and T_1 and T_2 be two estimated trees for the same leaf set. If T_1 is a refinement of T_2, then the False Negative rate of T_1 will be less than or equal to that of T_2, and the False Positive rate of T_1 will be at least that of T_2.*

This observation will turn out to be important in understanding the relationship between the error rates of consensus trees, and how they compare to the error rates of the trees on which they are based.

2.6 Rogue taxa.

Sometimes two trees are very different primarily (or even exclusively) in terms of how one leaf is placed. For example, in the evolutionary trees estimated for the Indo-European (IE) family of languages, Ringe *et al.* have noted that Albanian tends to "float" within the IE tree, fitting equally well into several places. (Such a taxon is called a "rogue taxon" in the biological literature.) Similarly, but for different reasons, Germanic can place differently within the IE tree, depending upon the choice of phylogenetic reconstruction method and whether lexical data alone or morphological and phonological characters are used.

The definition of "rogue taxon" was not precisely stated, and scientists can disagree as to what constitutes a rogue taxon. Furthermore, defining what a rogue taxon is depends on the technique used to estimate the tree on the taxa – and so the rogue taxa could be different depending on the method.

We will return to the problem of detecting rogue taxa later in the textbook, after we discuss phylogeny estimation methods.

2.7 Induced subtrees

A comparison of two trees that differ only in terms of the placement of a rogue taxon (e.g., Albanian within Indo-European) would best be done not through the use of FN and FP rates, but through other measures. To enable these more fine-tuned comparisons, we define the notion of “induced subtrees”. Later on we will talk about phylogeny reconstruction methods that operate by combining subtrees together, and there the concept of induced subtrees will also be helpful.

Suppose you have a tree T (rooted or unrooted), and a subset of the leaf set that is of particular interest to you, and you wish to know what T tells you about that subset. For example, T could be on a, b, c, d, e, f, but you are only interested in the relationship between the taxa a, b, c, d. To understand what T tells you about a, b, c, d, you do the following: delete the other leaves and their incident edges, and then suppress nodes of degree two. If A is the subset of interest, then we denote by $T|A$, the subtree of T induced by the set A. See Figure 18 for a tree and one of its induced subtrees.

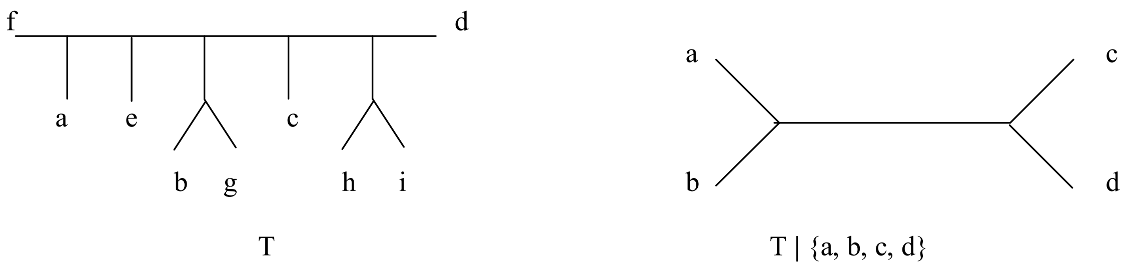


Fig. 18. Tree T and the subtree it induces on a, b, c, d

3 Constructing trees from subtrees

3.1 Constructing rooted trees from rooted triples

Here we present an algorithm for constructing a rooted tree from its set of "rooted triples", where by "rooted triple" we mean a rooted three-leaf tree. We indicate the rooted triple on a, b, c in which a and b are more closely related by $((a, b), c)$ (or any of its equivalents). We will also discuss the closely related problem of determining *whether* a set of rooted triplets is compatible with some tree, and constructing it if so.

Algorithm 1 for constructing trees from rooted triples. We consider the case where we are given a set $Trip$ of rooted triplets, with exactly one tree on every three species, and we want to construct the rooted tree T so that $Trip$ contains all the induced three-taxon trees in T.

To construct T from $Trip$, we do the following:

- If the number of taxa in $Trip$ is at most 3, just return the tree in $Trip$. Else:
    - Find a sibling pair a, b (that is, a pair of taxa that are always grouped together in any triple that involves them both).
    - Remove all rooted triples that include a from the set $Trip$, to produce a reduced set $Trip'$
    - Recursively compute a tree on $Trip'$ of rooted triples
    - Insert a into the tree by making it sibling to b.

Here we give an example of this algorithm. Suppose our input set $Trip$ has rooted triples

- $((a, b), c)$,
- $(a, (c, d))$,
- $((a, b), d)$, and
- $(b, (c, d))$.

We note that a and b are siblings, since any triple that involves them both puts them together. We remove all triples involving a from the set, and obtain set $Trip'$ with only one rooted triple: $(b, (c, d))$. We return this tree as the solution on $\{b, c, d\}$, and we insert a by making it a sibling to b. This gives us the tree $((a, b), (c, d))$, which agrees with every rooted triplet in the set above. Hence, the algorithm is correct on this input.

Now consider another input:

- $((a, b), c)$
- $((b, (c, d))$
- $((a, d), c)$
- $((a, b), d)$

What would happen if we try to run the algorithm on this input? We would detect that a and b are siblings, since they are never separated in any quartet in which they both appear, and we would remove one of them. Suppose we removed a. Then we remove all triplets that have a in them, and we reduce to a single triplet tree – $(b, (c, d))$. We add a as sibling to b and obtain the tree $((a, b), (c, d))$. However, this tree is inconsistent with the triplet $((a, d), c)$, and so the algorithm did not return a tree that agreed with all the rooted triplets.

The problem is that the set of rooted triplets isn't actually compatible - there is no tree that agrees with all the rooted triplets in the set, and the algorithm fails to detect this.

Therefore, we would need to modify it to determine *whether* the set $Trip$ was actually compatible with some tree. At a minimum, we should finish the algorithm by verifying that every rooted triplet is in agreement with the tree that is output. If so, then all is well; otherwise, the algorithm should indicate that the set of rooted triplets is not compatible with any tree.

On the other hand, there is still one more problem - if the set of rooted triplets is not compatible, then it is possible that there will not be any pair of leaves that pass the siblinghood test. Thus, the algorithm would need to be modified to allow for that possibility.

The final version of the algorithm that addresses these issues is as follows:

– If the number of taxa in $Trip$ is at most 3, just return the tree in $Trip$. Else:
  - Find a sibling pair a, b (that is, a pair of taxa that are always grouped together in any triple that involves them both). If no such pair exists, return "No compatibility tree exists". Else, continue.
  - Remove all rooted triples that include a from the set $Trip$, producing the reduced set $Trip'$
  - Recursively compute a tree on the reduced set $Trip'$ of rooted triples
  - Insert a into the tree by making it sibling to b.
  - Check that the resultant tree T^* agrees with every triplet in $Trip$. If so, return T^*, and otherwise return "No compatibility tree exists."

Note that this algorithm allows you to determine if a set of rooted triples are consistent with a tree, and to construct the tree when it is. However, it is only guaranteed correct when the set contains exactly one tree on every three taxa. For other cases, where - for example - the set $Trip$ might only contain trees on a proper subset of the triplets of taxa, another algorithm is needed.

Algorithm 2 for constructing trees from rooted triples. This second algorithm was developed by Aho, Sagiv, Szymanski, and Ullman in the context of relational databases. It's widely known in phylogenetics, though!

The input to this problem will be a pair, $(S, Trip)$, where S is a set of taxa, and $Trip$ is a set of triplet trees on S (with at most one tree for any three species).

– Group the set S of taxa into disjoint sets, by putting two leaves a and b in the same set if there is a rooted triple that puts them together. Compute the transitive closure of this relationship.

– If this produces one equivalence class, reject - no tree is possible. Otherwise, let $C_1, C_2, \ldots, C_k$ ($k \geq 2$) be the equivalence classes For each equivalence class C_i,
  - let $Trip_i$ be the set of triplets in $Trip$ that have all their leaves in C_i.
  - Recurse on $(C_i, Trip_i)$, and let t_i be the compatible rooted tree, if it exists.

  Make the roots of the $t_1, t_2, \ldots, t_k$ all children of a root, and return the final tree.

This surprisingly simple algorithm is provably correct, and runs in polynomial time. Proving that it works is also not that difficult.

3.2 Constructing trees from quartet subtrees

Just as rooted trees are defined by their rooted triples, an unrooted tree is defined by its unrooted quartet trees. For example, the tree given by $(1,(2,(3,(4,5))))$ is defined by the set of quartet trees $\{(12|34),(12|35),(12|45),(13|45),(23|45)\}$. It is also easy, algorithmically, to compute a tree T from the set $Q(T)$ of quartet subtrees. We call this the *All Quartets Method.*

All Quartets Method The input to this method is a set Q of quartet trees, with exactly one tree on every four leaves from a set S. We will assume that $|S| \geq 4$, since otherwise there are no quartets. We will also assume that every tree in Q is binary (fully resolved).

The algorithm we describe here addresses the case where it is possible for the quartet trees to be incompatible, so that no tree exists on which all the quartet trees agree.

- If $|S| = 4$, then return the tree in Q. Else, find a pair of taxa i, j which are always grouped together in any quartet that includes both i and j. If no such pair exists, **return** "No compatibility tree", and **exit**. Otherwise, remove i.
- Step 2: Recursively compute a tree T' on $S - \{i\}$.
- Step 3: Return the tree created by inserting i next to j in T'.

We show how to apply this algorithm on the input given above. Note that taxa 1 and 2 are always grouped together in all the quartets that contain them both, but so also are 4 and 5. On the other hand, no other pair of taxa are always grouped together. If we remove taxon 1, we are left with the single quartet on $2, 3, 4, 5$. The tree on that set is $(23|45)$. We then reintroduce the leaf for 1 as sibling to 2, and obtain the tree given by $(1, (2, (3, (4, 5))))$.

Note that the All Quartets Method suggests an algorithm for constructing trees: compute the tree on every quartet, and then combine them together. If all quartets are correctly computed, the resultant tree will be correct.

However, in practice, not all quartet trees are likely to be correctly computed, and so methods for constructing trees from quartet subtrees need to be able to handle some errors in the input trees.

Finding trees that satisfy a maximum number of the input quartet trees is an NP-hard problem, but when the set contains a tree on every quartet, then good approximation algorithms exist for this problem. On the other hand, when the set contains trees for only a subset of the possible quartets, then even determining if the set is compatible (i.e., can be combined together into a tree that agrees with all the quartets) is also NP-hard.

Heuristics for constructing trees that agree (possibly) with the largest number of quartet trees have been developed; see for example, the Quartet Puzzling algorithm of Strimmer and von Haeseler, Weight Optimization by Ranwez and Gascuel, and the Quartet Max Cut algorithm by Snir and Rao.

3.3 General supertree methods

The general context is that the input ("source") trees can be of arbitrary size (i.e., not all quartets, and not all triplets), and the objective is to put together the smaller trees into a tree on the entire set of taxa. As before, determining if all the subtrees are compatible (that is, if there is a tree on the entire set of taxa which agrees with all the smaller trees) is an NP-hard problem. Optimization problems in this area are therefore also NP-hard.

4 Constructing trees from qualitative characters

4.1 Introduction

We now turn to issues that relate to estimating trees from data. In essence, there is really one primary type of data used to construct trees – **characters**. An example of a character in biology might be the nucleotide (A, C, T, or G) that appears in a particular location within a gene, the number of legs (any positive integer), or whether the organism has hair (a boolean variable). In linguistics we find similar variety in characters; for example, it could be the cognate class for a semantic slot, whether or not a language has undergone a sound change, the particular way the language handles some aspect of its morphology, or the presence or absence of some typological features. In each of these cases, characters divide the dataset into different pieces, and the taxa (species or languages) within each piece are equivalent with respect to that character – they share the same **state**.

Mathematically, most models for the evolution of characters down trees assume that character state changes occur due to substitution. When the substitution process produces a state that already appears anywhere else in the tree, this is said to be **homoplastic** evolution (or, more simply, **homoplasy**). **Back-mutation** (reversal to a previous state) or **parallel evolution** (appearance of a state in two separate lineages) are the two types of homoplasy. When all substitutions create new states that do not appear anywhere in the tree, the evolution is said to be **homoplasy-free**. Furthermore, when the tree fits the character data so that no character evolves with any homoplasy, then the tree is called a **perfect phylogeny**.

Sometimes new states arise without replacement of the current state, so that a taxon exhibits two states (or more) at once. This is called *polymorphism*. Polymorphism in linguistic data occurs quite frequently – for example, when there are two or more words for the same basic meaning (examples include 'big' and 'large', or 'rock' and 'stone'). Long term polymorphism for linguistic characters does not seem to be tolerated well, so that over time, there are losses of character states, reducing the polymorphism load. Polymorphism in biology, however, can be quite commonplace, especially when considering different alleles for the same gene.

Finally, not all evolution is treelike, so that some characters can evolve with reticulation. For example, in linguistics, words can be borrowed (i.e., transmitted) between lineages. This can be frequent for lexical characters under some circumstances, perhaps also frequent for phonological characters, but unlikely for morphological features. In biology, horizontal gene transfer is common in some organisms (e.g., bacteria), and hybridization (whereby two species come together to make a new species) is also frequent for some organisms.

Molecular characters in biology are derived from alignments of nucleotide or amino acid sequences, and thus have a maximum number of possible "states" (four for DNA or RNA, and 20 for aminoacids). In linguistics, the number of possible states is two for presence/absence types of characters, but is otherwise *unbounded*. This is one of the implicit differences between linguistic characters and biological characters. Within a particular dataset, however, a character will exhibit only a finite number of states. When only two states are exhibited, the character is said to be **binary**. Some characters (for example, presence/absence characters) are explicitly always binary.

Finally, for some characters, there is an implicit and clear directionality of the evolutionary process. For example, presence/absence characters based upon sound changes (phonological characters) typically have an ancestral state (the absence of the sound change) and a derived state (the presence of the sound change).

Methods for estimating trees Methods for reconstructing trees from characters come in several variants. The most popular ones in linguistic phylogenetics are maximum compatibility (and its weighted variant) and maximum parsimony. These are two optimization problems that are closely related, and often have the same optimal solutions. However, methods based upon statistical models of evolution, and hence involving calculations of the **likelihood**, are also favored by some researchers. Finally, methods that first transform the character data input into distance matrices are also popular; these are called "distance-based" methods. We cover maximum compatibility and maximum parsimony in this section, and cover distance-based methods and methods based upon likelihood (both Maximum Likelihood and Bayesian methods) in later sections. Finally, these methods all produce trees, but reticulations (borrowing or creolization) can also occur. We will therefore include a chapter on estimating reticulate evolutionary histories.

Suppose we have n taxa, $s_1, s_2, \ldots, s_n$ described by k characters, $c_1, c_2, \ldots, c_k$. This input is typically provided in matrix format, with the taxa occupying rows and different characters occupying the columns. In this case, the entry M_{ij} is the state of the taxon s_i for character c_j. We can also represent this input by just giving the k-tuple representation for each taxon.

4.2 Constructing rooted trees from directed binary characters

Constructing trees from characters can be very simple or very complicated. We begin with the very simple situation: binary characters that evolve without any *homoplasy*. To make it even simpler, we'll assume that the characters are given with an orientation, so that the ancestral state is known. We call these "directed binary characters".

Many of the linguistic trees have been constructed using directed binary characters that evolve without homoplasy. In this case, you can identify the derived state for each character, and since the evolution is without homoplasy, the languages that exhibit the derived state must form a clade in the true tree. Therefore, the problem becomes quite simple: *given a set of clades in a tree, construct the tree.* This is a problem we solved in the previous section! Note that this produces a rooted tree, with the root having the ancestral state for all the characters.

4.3 Constructing unrooted trees from compatible binary characters

The problem becomes slightly harder when we are given binary characters without information on the ancestral state. However, this case also has an easy solution, as we will show. We treat one taxon as the root, and let its state for each character be the ancestral state of that character. This makes the problem equivalent to constructing a rooted tree (on the remaining taxa) from clades. Once that rooted tree is constructed, we add the taxon that represented the root to the tree, and then *unroot* the tree.

In the case where the characters evolve without homoplasy (and so are compatible on a tree), whether the characters are directed or not, the minimum tree that fits the character evolution assumptions is *unique*, and can be computed in polynomial time. Here, by minimum we mean that we seek a tree in which no edge can be contracted while still having the property that all the characters are compatible. This minimal tree may not be binary, however, since the tree that is computed will only have edges on which the binary characters change. More generally, what this means is that if you use only a subset of the available characters, the tree you obtain may not be fully resolved. Importantly, this means that the interpretation of polytomies (nodes of high degree) in trees estimated using this technique is that they are likely due to incomplete information (not all the characters are used, or perhaps are not available).

Note that the algorithms can be used in two ways: to construct a tree for which the characters evolve without homoplasy, or to determine that no such tree exists!

Finally, note that these algorithms require that all taxa exhibit states for all the characters – that is, it is not possible to apply the algorithms when some character data are *missing*. Therefore, when the state of some characters for some taxa is unknown, you cannot use these algorithms.

We now turn to some examples.

Example: Suppose that the input is given by

- $A = (1, 0, 0, 0, 1)$
- $B = (1, 0, 0, 0, 0)$
- $C = (1, 0, 0, 1, 0)$
- $D = (0, 0, 0, 0, 0)$
- $E = (0, 1, 0, 0, 0)$
- $F = (0, 1, 1, 0, 0)$

In this case, there are two non-trivial characters (defined by the first and second positions), but the third through fifth positions define trivial characters. When we apply this algorithm, we pick one taxon as the root. Since the choice of root doesn't matter, we'll pick A as the root. The clades under this rooting are: $\{D, E, F\}$ (for the first character), and $\{E, F\}$ (for the second character). Also, under this rooting, A, B and C are identical (since we only consider the non-trivial characters). The tree we obtain for the remaining taxa, using the algorithm on clades, is $(D, (E, F))$. Adding in A, B, C as the root, we obtain the unrooted tree $(A, B, C, (D, (E, F)))$.

Comments: Many comments are worth making now. First, note that when the dataset consists of binary characters that evolve without homoplasy a unique minimal tree will exist, but it may not be *binary*. That is, it may not be fully resolved. Only those edges of the true tree on which changes occur will be reconstructed. Therefore, if you take only a subset of the characters and apply the algorithm, you may construct an incompletely resolved tree. In this case, the proper interpretation of the polytomies is that you *lack* information sufficient to resolve the tree.

4.4 General issues in constructing trees from characters

Until this point, our discussion has assumed that all taxa exhibit states for all characters in the input matrix, and that all the characters are compatible and binary (exhibit two states). Under these assumptions, it is easy (polynomial time, and easy to do by hand) to construct trees: we use the algorithm for constructing trees from compatible bipartitions, and use the tree that results. However, can we apply the simple algorithm when these assumptions do not hold? That is, when the input consists of characters that are binary (for example, presence/absence), but we are missing some information? Or when the input is non-binary? Or when the input is incompatible?

Missing data issues We begin with the complication when not all taxa exhibit states for all the characters. A natural approach to take is find out if it is possible to assign values for the missing entries in the character matrix so as to make the input compatible. See, for example, the following input, where "?" means that the state is unknown.

- $A = (0, 0, 0)$
- $B = (0, 1, 1)$
- $C = (1, ?, 1)$
- $D = (1, 0, ?)$
- $E = (?, 0, 0)$

This input does admit assignments of states to the missing values, so as to produce a compatible data matrix:

- $A = (0, 0, 0)$
- $B = (0, 1, 1)$

– $C = (1, 0, 1)$
– $D = (1, 0, 1)$
– $E = (0, 0, 0)$

We know this is compatible, because the tree given by $(A, (E, (B, (C, D))))$ is compatible with these characters (i.e., it is a perfect phylogeny).

By contrast, there is no way to set the values for the missing entries in the following matrix, in order to produce a tree on which all the characters are compatible:

– $A = (0, 0, ?)$
– $B = (0, 1, 0)$
– $C = (1, 0, 0)$
– $D = (1, ?, 1)$
– $E = (?, 1, 1)$

Figuring out that these characters are incompatible, no matter how you set the missing data, is not that trivial. But as there are only three missing values, you can try all $2^3 = 8$ possibilities. More generally, however, answering whether an input with missing data admits a perfect phylogeny is NP-hard, even when only two states otherwise appear. The computational method for solving this problem involves a mathematical transformation of the input matrix so that there are no missing entries. Instead, every question mark is replaced with a new state that does not appear in the dataset for any other language. Thus, the initial data matrix might only have two states (presence/absence, or 0/1), but the transformed data matrix could have many more states. For example, if we apply this technique to the input given above, we obtain:

– $A = (0, 0, 2)$
– $B = (0, 1, 0)$
– $C = (1, 0, 0)$
– $D = (1, 2, 1)$
– $E = (2, 1, 1)$

Now, if we begin with an input M with missing entries, and do this transformation, we obtain a new input M'. Note that a perfect phylogeny exists for M if and only if a perfect phylogeny exists for M'. Unfortunately, while determining if a perfect phylogeny exists for binary characters is easy (and can be constructed in polynomial time), determining if a perfect phylogeny exists for multi-state characters is computationally harder: no longer polynomial time, and not easy to do by hand.

Constructing trees from compatible multi-state characters The previous section was all about binary characters, typically based upon presence/absence of some feature. We also primarily focused on characters that evolve without homoplasy (back-mutation or parallel evolution). But what about other types of characters? Lexical characters and morphological characters are likely to have more than two states in many language families, for example. How do we construct trees from multi-state characters?

We begin with the assumption that the characters evolve without homoplasy. In this case, algorithms to find the trees on which all the characters evolve without any homoplasy *do* exist, but they are computationally more expensive – no longer polynomial, as in the case of binary characters. Also, it is no longer the case that there is a unique minimal tree which is consistent with the input!

Before we go into how to construct trees from multi-state characters, we address the “simpler” issue of testing whether a multi-state character is “compatible” on a tree (meaning, it could have evolved without any homoplasy on the tree).

Testing compatibility of a character on a tree To do this, we wish to set states of the character for the internal nodes of the tree in such a way that for each state of the character, the nodes of the tree that exhibit that state are connected. When this is the case, the character is said to be **compatible** with the tree. Testing whether a character is compatible on a tree is straightforward, and can be done by eye.

For a given internal node v in the tree, if it lies on a path between two leaves having the same state x, we assign state x to node v. If this assignment doesn't have any conflicts – that is, as long as we don't try to assign two different states to the same node, then the character can evolve without any homoplasy on the tree. Otherwise, we either have to posit homoplasy (back mutation or parallel evolution) or *polymorphism* – the presence of two or more states at some node.

It is evident that not all linguistic characters evolve without homoplasy, and so when a character is incompatible with a tree, the linguist must determine the best explanation: is it likely that the character evolved with homoplasy, perhaps with borrowing, or is the problem perhaps that the tree is incorrect? Answering this depends upon linguistic judgments!

4.5 Maximum compatibility

Algorithms for constructing trees under the assumption that all the characters are compatible will fail when any character evolves with homoplasy. While characters are rarely likely to evolve without homoplasy in biological datasets, the assumption of homoplasy-free is more realistic in linguistics, for a number of reasons. However, not all linguistic characters are homoplasy-free! For example, sound changes (and typological features) can be so natural that they appear in many lineages, and so evolve with a lot of parallel evolution. Lexical characters can evolve with borrowing (and hence not be compatible on the underlying genetic tree), and thus require networks (rather than trees) to properly represent their history. Finally, semantic shift can result in lexical characters that are not compatible with the genetic tree.

Therefore, when given a set of binary characters for a linguistic group, if the algorithm for homoplasy-free evolution does not produce a tree, the linguist's task is to come up with a reasonable explanation for the character evolution, and identify the characters most likely to have evolved with homoplasy. Removing those characters, and reapplying the algorithm, can be used to good advantage.

The process of identifying and removing problematic characters, and then repeating the phylogenetic analysis makes sense from a linguistic point of view, but presents several challenges. First, sometimes the dataset is large, making this process a potentially very long one. Second, the identification of problematic characters in itself involves a great deal of expertise, and unless the identification of these characters is based upon solid linguistic grounds, the removal of problematic characters may simply lead to reinforcement of the linguist's biases.

Automated techniques to identify and remove characters from datasets so as to produce compatible sets of characters do exist, however, and are the subject of this next section.

We begin with the definition of the maximum compatibility problem. Recall that a character c is said to be compatible on a tree T if it is possible to define the character states at the internal nodes so that for all states of c, the set of nodes exhibiting that state is connected. An equivalent definition is that if c exhibits r states on the tree T, then there are exactly $r-1$ edges of the tree T on which c changes state.

The maximum compatibility problem is then as follows:

Maximum Compatibility

Input: Character matrix M with n rows and k columns (so that M_{ij} is the state of taxon s_i for character c_j)

Output: Tree T on the leaf set $S = \{s_1, s_2, \ldots, s_n\}$ on which the number of characters in $C = \{c_1, c_2, \ldots, c_k\}$ that are compatible is maximized.

Related to this search problem is the problem of determining the number of characters that are compatible on a given tree (i.e., scoring a tree with respect to compatibility).

Computing the compatibility score of a tree

Input: Character matrix M with n rows and k columns (so that M_{ij} is the state of taxon s_i for character c_j), and a tree T with leaves labelled by the different species, $s_1, s_2, \ldots, s_n$.
Output: The number of characters that are compatible on T.

This problem is polynomial, since (as we showed in the previous section), determining if a character is compatible on a tree can be done quite simply. Hence, scoring a given tree under compatibility is polynomial.

On the other hand, finding the tree with the largest compatibility score is more computationally challenging. If we use an exhaustive search technique, scoring each of the possible solutions in turn, this would take time $O(t(n)nk)$, where n is the number of taxa, k is the number of characters, and $t(n)$ is the number of binary trees on n leaves ($t(n) = (2n-5) \cdot (2n-7) \cdot 3$.) The reason we only need to examine binary trees, is that optimal solutions to maximum compatibility are obtained at the binary trees (i.e., if a non-binary tree could be any optimal solution, each of its refinements will also be an optimal solution).

We now look at computing solutions to maximum compatibility. On the input below, all the characters are compatible, and the solution would be the tree T on which all the characters are compatible.

- $A = (0, 0, 0)$
- $B = (0, 0, 3)$
- $C = (1, 1, 0)$
- $D = (1, 1, 1)$
- $E = (2, 1, 0)$
- $F = (2, 2, 4)$

One tree on which these characters are all compatible is given by $(A, (B, (E, (F, (C, D)))))$.

On the next example, however, the set of characters is not compatible, and the best solution(s) would have only two characters that are compatible.

- $A = (0, 1, 0)$
- $B = (0, 0, 0)$
- $C = (1, 0, 0)$
- $D = (1, 1, 1)$

Note that the third character is compatible on every tree, but the first two characters are incompatible with each other. Therefore, any tree can have at most one of these first two characters compatible with it. One of those trees is given by $((A, B), (C, D))$, and the other is $((A, D), (B, C))$. The third possible unrooted tree on these taxa is $((A, C), (B, D))$, which is incompatible with both these characters.

We now consider this problem for linguistic phylogenetics. In this case, the use of maximum compatibility is motivated by the idea that properly selected and coded characters ought to be compatible on the true tree, assuming there *is* a true tree (as opposed to a network in which taxa evolve with borrowing as well as with genetic descent). This idea follows from the selection of characters that are unlikely to evolve with homoplasy. And while all characters can exhibit homoplasy, especially if there are mistakes in character encoding (that is, the assignment of character states), some characters are less likely than others. Thus, *maximum weighted compatibility* is also a relevant optimization problem in linguistic phylogenetics:

Maximum weighted compatibility

Input: Matrix M as above, but with characters given with positive weights, $c_1, c_2, \ldots, c_k$.
Output: Tree T on the set of taxa so as to maximize the sum of the weights of the compatible characters on T.

It is clear that the assessment of the relative probability of homoplasy involves a great deal of linguistic expertise and, of course, personal opinion. Thus, assigning weights to characters is best done by a linguist skilled in the language family. (Assigning states to taxa for different characters also takes linguistic expertise, for that matter!)

As with Maximum Compatibility, weighted maximum compatibility is optimized on binary trees. Thus, any heuristic for solving weighted maximum compatibility need only examine completely resolved trees.

Finding a solution to maximum compatibility (whether weighted or unweighted) is hard, because the problem is NP-hard. Thus, solutions that are guaranteed to solve the problem optimally use techniques like branch-and-bound or exhaustive search. Unfortunately, no software exists for solving this problem in an automated fashion. Instead, solutions to this problem are obtained by first finding solutions to maximum parsimony (discussed below), and then scoring each of the trees with respect to the maximum compatibility criterion. This approach works reasonably because the two problems are very similar, so that optimal solutions to one problem are often near-optimal solutions to the other. Furthermore, while effective software for maximum compatibility does not really exist, there are many very effective software packages for maximum parsimony, due to its frequency of use in biological phylogenetics. In the next section, we define the maximum parsimony problem, and discuss software used to solve this problem.

4.6 Maximum Parsimony

Maximum parsimony is an optimization problem in which a tree is sought for an input character matrix (the same type of input as is provided to maximum compatibility), for which the total number of character state changes is minimized. We begin this discussion by making a precise statement of what is meant by the number of state changes of a character on a tree.

For those characters that evolve without any homoplasy, it is easy to assign states on the tree so that the character changes state the minimum number of times. And, in fact, if the character exhibits r states on the dataset, then it will change state exactly $r-1$ times if it evolves without homoplasy (and otherwise it will change state more than $r-1$ times). Determining the minimum number of times the character must change state is a polynomial time problem, but not an easy one to do by hand. We will return to this another time! However, on small enough trees, it can be done by eye if you are careful.

Recall the discussion of this issue given in the introduction. First, if a character is defined for all nodes in a tree, then this means that every node of the tree is given a state for that character. In this case, the number of state changes for that character on the tree is simply the number of edges on which the character changes state, and is easily computed. However, if the character is only defined on the leaves, we will want to compute the *best* state assignment to the internal nodes so as to minimize the total number of state changes for the character. This problem is easily done by inspection for small trees, and can even be done efficiently (meaning in polynomial time) on large trees – although the technique is then best done using software rather than by eye. Thus, when the tree T and character matrix M are given, it is possible to compute the number of character state changes on T for the matrix M in polynomial time. This minimum total number of changes of a character matrix M on a tree T is called the *length* of the tree, and also the *parsimony score*. Thus, maximizing parsimony means producing the minimum parsimony score. Somewhat confusing terminology, eh?

Finding the best tree T for a given character matrix M is the maximum parsimony problem, i.e.:

Maximum parsimony

Input: Matrix M with n rows and k columns, where M_{ij} denotes the state of taxon s_i for character c_j.
Output: Tree T on leaf set $\{s_1, s_2, \ldots, s_n\}$ with the smallest total number of changes for character set $\{c_1, c_2, \ldots, c_k\}$.

While maximum parsimony is polynomial time if the tree is given, the problem is NP-hard when the tree is not known and must be found. Furthermore, exhaustive search or branch-and-bound solutions are limited to small datasets. Fortunately, effective search heuristics exist which enable reasonable analyses on large datasets (with hundreds or even thousands of taxa). These heuristics are not *guaranteed* to solve the optimization problem exactly, but seem to produce trees that are close in score and topology to the optimal solution, in reasonable timeframes (i.e., hours rather than months).

Like Maximum Compatibility, Maximum Parsimony is optimized on binary trees, and heuristics for solving maximum parsimony need only examine completely resolved trees. Even so, these heuristics are computationally expensive, taking (in some cases) many days of analysis to come to what can only be guaranteed to be local optima.

Scoring trees under Unweighted Maximum Parsimony We begin with the problem of computing the unweighted parsimony score of a fixed tree. In this problem, the tree is given and all substitutions have equal weight. We will show a very simple dynamic programming algorithm for this problem that allows you to compute the parsimony score in polynomial time. The algorithm also allows you to compute an assignment of states for each character to each node in the tree, in such a way that you produce the smallest number of changes.

The simplest form of the algorithm operates as follows (here we assume the input tree is unrooted and binary; modifying the algorithm for non-binary trees is slightly more complicated): The algorithm is applied to each character independently.

- Root the tree on an edge, thus producing a rooted binary tree.
- If x is a leaf, let $A(x)$ denote the state at the leaf x for the given character.
- Starting at the nodes v which have only leaves as children, and moving up the tree (towards the root), do the following:
  - If v has children w and x, and if $A(w) \cap A(x) \neq \emptyset$, then set $A(v) = A(w) \cap A(x)$. Else, set $A(v) = A(w) \cup A(x)$.
- When you reach the root, r, pick an arbitrary state in $A(r)$ to be its state. Then traverse the tree downwards towards the leaves, picking states for each node, as follows:
  - If the parent of node y has been assigned a state that is within $A(y)$, then set the state for y to the same state as was assigned to its parent. Otherwise, pick an arbitrary element in $A(y)$ to be its state.

At the end of this two-phase process (one up the tree, and one down), you will have assigned states to each node in the tree. Note that in the upwards phase, some nodes will be assigned definite states, but others may be given more than one possible state. When $A(v)$ has only one element in it, then v' assigned under maximum parsimony is uniquely determined. This will be relevant to issues involving estimation of the properties of ancestral taxa, using maximum parsimony.

Scoring a tree under weighted maximum parsimony. The problem is somewhat more complicated if the substitution costs can depend on the particular pair of letters. For example, there are two different types of nucleotides – purines (which are A and G) and pyrimidines (which are C and T). Substitutions that change a purine into a purine, or a pyrimidine into a pyrimidine (which are called "transitions") are considered more likely than substitutions that change a purine into a pyrimidine or vice-versa (which are called "transversions"). Therefore, one variant of maximum parsimony would treat these two types of substitutions differently, so that transitions would have lower cost than transversions, but any two transitions or any two transversions would have the same cost.

More generally, suppose you have an alphabet Σ with r letters, and so you represent the substitution cost as a symmetric rxr matrix M, where $M[x, y]$ is the cost of substituting x by y. Clearly $M[x, x]$ should be 0. If all entries off the diagonal are the same, then this is identical to unweighted maximum parsimony, and the previous algorithm works. But what if the entries off the diagonal are different?

As it turns out, this is not really any harder than unweighted maximum parsimony. Let t be an unrooted binary tree with leaves labelled by sequences of length k, all drawn from Σ^k. We root t on an edge, thus producing a rooted binary tree T, in which only the leaves are labelled by sequences. We consider a single character (site) at a time.

We define the following variables:

- For every vertex v in T, we let $A(v)$ denote the state at v. Thus, $A(v)$ is defined by the input for each leaf v, but will be set during the algorithm for the remaining nodes.
- For every vertex v in the rooted tree T, and for every letter x in Σ, we define $Cost(v, x)$ to be the *minimum parsimony cost of the subtree T_v over all possible labels at the internal nodes of T_v, given that we label v by x.*

How do we set $Cost(v, x)$? If v is a leaf, then we set $Cost(v, x) = 0$ if $A(v) = x$, and otherwise we set $Cost(v, x) = \infty$. Then, if v is a node that is not a leaf, and if we have already computed $Cost(w, x)$ for all nodes w in the subtree below v, and for all letters x in Σ, we can compute $Cost(v, x)$ as follows. Let w and w' be the two children of v. Then:

$$Cost(v, x) = \min\{Cost(w, y) + M[x, y] : y \in \Sigma\} + \min\{Cost(w', y) + M[x, y] : y \in \Sigma\}$$

To see how this works, suppose v has two children w and w' and they are both leaves. In this case, $Cost(w, x) = \infty$ if $A(w) \neq x$, and otherwise $Cost(w, x) = 0$. Hence, $Cost(v, x) = M[x, A(w)] + M[x, A(w')]$, which is what we want.

Now consider the case where one child w of v is a leaf and the other w' is not. Then $Cost(v, x) = M[x, A(w)] + \min\{Cost[w', y] + M[x, y] : y \in \Sigma\}$. What this means is that the smallest cost you can get for the tree T_v given that you label v by x is obtained for *some* way of labelling the child w' with a letter in Σ. Suppose y is the best way of labelling w', given that we've constrained the label at v to be x. Now consider the total cost of the entire subtree T_v; this is computed by summing the costs on the edges. The cost of the edge (v, w) is simply $M[x, A(w)]$, and the cost of the edge (v, w') is simply $M[x, A(w')] = M[x, y]$. The sum of the costs of the edges in the subtree $T_{w'}$ is then $Cost(w', y)$, since we've said that the label at w' is y.

The case where both children of v are not leaves can be analyzed similarly, showing that the formula is correct.

Therefore, the algorithm would compute $Cost(v, x)$ for all nodes v and all letters x as you go from the bottom of the tree up to the root. Therefore, you should not calculate $Cost(v, x)$ until you have calculated $Cost(w, y)$ for all nodes w below x and all letters $y \in \Sigma$. To determine the parsimony score of the tree, you calculate $\min\{Cost(r, x) : x \in \Sigma\}$, where r is the root of the tree.

An optimal label at the root r will be x_0 such that $Cost(r, x_0)$ is the parsimony score of the tree. However, to label the remaining nodes, you will need some additional calculations.

Suppose that as you go up the tree, calculating $Cost(v, x)$ for each node v and letter x, you record at least one pair of values for y and y' such that $Cost(v, x) = Cost(w, y) + Cost(w', y') + M[v, y] + M[v, y']$. Then, to set the optimal labels for the internal nodes of the rooted tree T, you first set the label x_0 for the root r. Then, you visit the two children w, w' of r. Since you have recorded the pair of values y and y' associated to $Cost(r, x_0)$, you set $A[w] = y$ and $A[w'] = y'$. Having set these labels, you can then continue down the tree and set the labels for every internal node.

Thus, the second phase of the algorithm in which you set the labels at the internal nodes can be performed in $O(n)$ time, where n is the number of leaves in the tree, provided that during the first phase you have recorded the additional information.

4.7 Binary encoding of multi-state characters

One of the ways that some researchers have tried to estimate trees from multi-state characters is by doing a "binary encoding" of the character. This results in a replacement of the original multi-state character matrix by a binary character matrix, which can then be analyzed using techniques that require inputs to be binary. The advantage of this is that it enables the use of technologies that cannot be run on multi-state characters. However, there are definite disadvantages, which will be discussed later.

The technique is as follows. Suppose you have a character which exhibits r states on a set S of taxa. You replace that single r-state character by r binary characters, one for each state. Then, the character for the state i will indicate whether the language has that state or not. For example, consider a three-state character C defined on set $\{L_1, L_2, ..., L_6\}$, so that $\{L_1, L_2\}$ have state 1, $\{L_3, L_4\}$ have state 2, and $\{L_5, L_6\}$ have state 3. The binary encoding of this three state character would produce three binary characters. The character for state 1 would split the taxa into two sets: those having state 1 (i.e., $\{L_1, L_2\}$) and those not having state 1 (i.e., $\{L_3, L_4, L_5, L_6\}$. Note that the evolution of character C might have very different properties than the evolution of the binary characters derived from C. For example, this character C is compatible on the tree $((L_1, L_2), (L_3, (L_4, (L_5, L_6))))$, but not all its derived characters are. Also, C will change state on some edges of the tree but not all its derived characters will.

4.8 Informative and uninformative characters

In terms of solving maximum parsimony, or analyzing the properties of the maximum parsimony as a method, it is helpful to evaluate when a character has an impact on the tree that will be returned. In other words, if your input matrix M (where $M_{i,j}$ is the state of the taxon s_i for character j) is given as input, you would like to know whether removing some specific character (say character x) has any impact on the tree that is returned. Since removing a character amounts to removing one column in the matrix, this would be the same as saying "If we define matrix M-x to be the matrix obtained by taking M and removing column x, when is it guaranteed that MP(M-x) returns the same optimal tree (or set of optimal trees) as MP(M)?" A character that has *no impact* on tree estimation using maximum parsimony methods (when solving MP exactly) is called "parsimony uninformative", and is formally defined as follows:

Definition of Parsimony Uninformative. Let x be a character defined on set S of species. Then x is parsimony uninformative if, for all matrices M for S the set of optimal parsimony trees on M is identical to the set of optimal parsimony trees on $M + x$, where $M + x$ denotes the matrix obtained by adding column x.

As a consequence, the set of optimal parsimony trees will not change by removing a parsimony uninformative site from any alignment. All other characters are called "parsimony informative". Removing parsimony uninformative characters can result in a speed-up in the search for optimal trees (especially if there are many such characters). Equally importantly, thinking about which characters are parsimony informative or not will help you understand the different impact of different characters on phylogeny estimation using maximum parsimony.

The same property can be asked about *any* phylogeny estimation method, obviously, and so we can ask whether a character is "compatibility-informative".

It is not hard to see the following:

Lemma 3. *A character is parsimony-informative and "compatibility-informative" if and only if it has at least two "big states", where a state is "big" if it has at least two taxa in it.*

Proof. This is left to the reader.

Other methods can be used to estimate trees, of course, and so the definition of what constitutes "informative" has to be based on the method. Thus, later in this text we will discuss another character-based method, maximum likelihood. At that point you should consider which characters will be informative for maximum likelihood.

5 Distance-based methods

We now turn to a discussion of phylogeny estimation from distances. That is, the input is a matrix of pairwise distances for the set of taxa, and the algorithm computes a tree (often with branch lengths, and sometimes with a root) using the input distance matrix. While these methods are based upon distances, almost all of the techniques used to compute distances between taxa begin with the same character data used in character-based estimations. Thus, distance-based methods are in a fundamental sense also character based.

5.1 Step 1: computing distances

The first step in a distance-based method is producing the pairwise distance matrix. In linguistics, these distances are typically based upon word lists, where the distance between two taxa is just the number of words that they are not cognate for. In this setting, two languages have distance 0 if they share cognates for all the semantic slots in the word lists. These distances are often normalized by the number of semantic slots in the word list, to produce distances that lie between 0 and 1. The resultant distance is called the **normalized Hamming distance**, or the **p-distance**. Alternatively, the pairwise distance between languages can be based upon an edit distance between the written or spoken forms of a word. In this scenario, two words that are cognate can still contribute to the distance between two languages, and there is no need to determine cognacy.

In biology, while uncorrected distances (i.e., p-distances) are also sometimes used, more typically these distances are corrected with respect to an assumed stochastic model of evolution. These "corrected" distances attempt to account for hidden changes (e.g., if you observe A in one string, and C in another, it is possible that more than one substitution occurred in that position in the history between those two strings).

The output of this first step is a matrix of pairwise distances. This matrix will be 0 on the diagonal and symmetric, but may not satisfy the triangle inequality (that is, it may be that for some triple of taxa, s_i, s_j and s_k, we have $d(s_i, s_j) + d(s_j, s_k) < d(s_i, s_k)$). In fact, with the corrected distances calculated under the various stochastic models of evolution, the assumption of the triangle inequality will generally not hold.

5.2 Step 2: computing a tree from a distance matrix

There are many methods used to construct trees from distance matrices. Here we describe a few.

UPGMA The standard approach used in linguistics is a variant of UPGMA, which stands for "Unweighted Pair Grouping Method of Agglomeration". In this kind of method, the tree is constructed by picking the pair of taxa that have the smallest distance, and making them siblings. One of them is then removed (or else both are removed and replaced by a new taxon), and the process is repeated. This technique produces a rooted tree, but the root can be ignored.

We begin with an example of UPGMA, applied to a case where the distances obey a strong clock. Figure 19 gives an ultrametric matrix, and Figure 20 gives the rooted tree realizing that matrix.

Other distance-based methods Not all distances obey a strong clock, and UPGMA can fail when the input matrix is not sufficiently clocklike. Consider, for example, the tree given in Figure 21. This tree has lengths on each edge, and thus defines a distance between every pair of leaves obtained by adding up the lengths of each edge. Note that the pair that minimizes the distance is L_1, L_2, but that these are not siblings! Thus, when UPGMA is applied to the matrix for this tree, it will produce the *wrong* tree.

	B	C	D
A	2	16	16
B		16	16
C			10
D			

Fig. 19. Ultrametric matrix

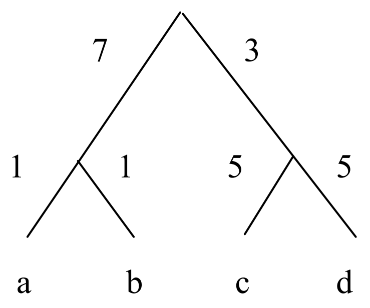
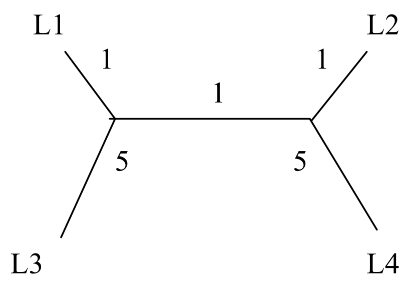


Fig. 20. Tree realizing the ultrametric matrix from Figure 19



	L2	L3	L4
L1	3	6	7
L2		7	6
L3			11
L4			

Fig. 21. Additive matrix and its edge-weighted tree

When a distance matrix $M_{i,j}$ fits an edge-weighted tree T exactly, in that the distance $M_{x,y}$ equals the path distance (sum of edge weights) in the tree T exactly, the matrix is said to be **additive**. Furthermore, given any additive matrix, there is a unique tree (if no zero-length edges are permitted) which fits the matrix exactly. A rather beautiful theorem can be stated about additive matrices:

The Four Point Condition A nxn matrix A is additive if and only if for all four indices i, j, k, l, the median and largest of the following three values are the same:

- $A_{i,j} + A_{k,l}$
- $A_{i,k} + A_{j,l}$
- $A_{i,l} + A_{j,k}$

One direction of the proof of this theorem is easy–if the matrix is additive, then there is a tree T with positive edge weights w realizing the additive matrix. In that case, given the four indices, i, j, k, l, you can assume (without loss of generality) that the tree has an edge separating i, j from k, l. In that case, $A_{ij} + A_{k,l}$ is smaller than the other two sums, and these other two sums have the same total score.

Given this characterization of additive matrices, it is easy to see that a tree can be produced from its additive matrix (and we present one such below). Note, also, that additive distance matrices do not have to reflect a clock – therefore, it *is* possible to construct trees using distance-based methods even when taxa do not evolve in a clocklike fashion.

However, what about estimating trees from distance matrices that are not additive? Here we show a method that can always estimate a tree on four-leaf datasets, whether or not the matrix is additive. It is based upon the Four-Point Condition, and so will be guaranteed to return the correct tree when the matrix is additive for that tree.

Estimating four-leaf trees: The Four-Point Method Given four taxa, A, B, C, and D, and given the distance matrix M on the four taxa, group the taxa into two sets of two taxa each, so as to minimize the pairwise sum. That is, compare $M(A, B) + M(C, D), M(A, C) + M(B, D)$, and $M(A, D) + M(B, C)$. If $M(A, B) + M(C, D)$ is minimum among these three pairwise sums, then return the tree $((A, B), (C, D))$. (Similarly for the other results.)

Estimating larger trees Among the methods that construct trees from distances, neighbor joining and FastME are among the more accurate (in the sense that the trees they produce are closer to the true trees than most other distance based methods). Furthermore, even when the evolution is somewhat clocklike, using methods like neighbor joining and FastME tends to produce better (more accurate) trees than using UPGMA. However, when the assumption of clocklike evolution is dropped, locating the root is very difficult. For this reason, distance-based methods generally do not produce rooted trees, but rather just unrooted trees. Finally, when the matrix is "close enough" (in a precisely quantified manner) to additive, then estimations of the tree can still be guaranteed to be correct, but estimations of the branch lengths will not be guaranteed to be correct.

The Naive Quartet Method. We now describe a very simple algorithm for estimating trees. It is so simple, it can be applied by hand! It uses the Four Point Method to compute trees on every four taxa, and then combines the trees into a tree on the full set of taxa using the *All Quartets Method* described in Section 3.2.

It is actually easy to see that if the matrix M is additive the Four Point Method will be correct on every four taxa, and so the quartet trees that are produced will be correct. Combining quartet trees into a tree on the full dataset is then straightforward (see the section on computing trees from quartet trees). However, here too if the input matrix is not additive, then some quartet trees can be incorrect, and the combination of these quartet trees into a tree on the full set of taxa can be problematic.

6 Statistical methods of phylogeny estimation

Phylogeny estimation is often posed as a statistical inference problem, where the taxa evolve down a tree via a probabilistic process. Statistical estimation methods take advantage of what is known (or hypothesized) about that probabilistic process in order to produce an estimate of the evolutionary history. That estimate can include a range of hypotheses – starting with the underlying tree, and perhaps also the location of the root, the time at the internal diversification events, the rates of evolution on each branch of the tree, etc. When we consider phylogeny reconstruction methods as statistical estimation methods, many statistical performance issues arise. For example: is the method guaranteed to construct the true tree (with high probability) if there is enough data? How much data does the method need to obtain the true tree with high probability? Is the method still relatively accurate if the assumptions of the model do not apply to the data that are used to estimate the tree?

In order to understand these questions, we will begin with some simpler problems that can also be posed as statistical estimation problems.

6.1 Introduction to Markov models of evolution

Markov models of evolution form the basis of most computational methods of analysis used in phylogenetics, and can be used to describe how qualitative characters with any number of states evolve. The simplest of these are for two states, reflecting the presence or absence of a trait. But more commonly, these models are used for nucleotide or amino-acid sequences, and so have 4 or 20 states, depending on the type of data. They can also be used (less commonly) for codon models, in which case they have 64 states.

The mathematics for these models is similar, independent of the number of states. So, we'll start by discussing a simple model for two states, the Cavender-Farris model.

Cavender-Farris model. Under the Cavender-Farris model, the probability of absent or present is the same at the root, and that this can change on the edges of the tree. To govern the changes, we associate a parameter $p(e)$ to every edge e in the tree, where $p(e)$ denotes the probability of changing state (from absent to present, or vice-versa). The model also requires that $0 < p(e) < 0.5$.

Note that this model only describes how a single state evolves. Therefore, to extend this model so that it is applicable to analyses of many two-state characters, we need to specify how different characters can differ in their evolutionary parameters. In the simplest form, the assumption is that all characters have exactly the same parameter values, and that the evolve independently. This is generally called the *i.i.d.* (identical and independently distributed) assumption.

DNA models. Because there are four nucleotides, A,C,T, and G, 4-state Markov models are used to describe nucleotide sequence evolution. The Jukes-Cantor model is the 4-state version of Cavender-Farris, with $0 < p(e) < 0.25$ defined for every edge, and all nucleotide substitutions have equal probability. Other more general 4-state models, such as the General Time Reversible model, are more commonly used.

Statistical Identifiability. Statistical identifiability is an important concept related to Markov models. We say that a parameter (such as the tree topology) of the Markov model is *identifiable* if, given the probability distribution of each character of the patterns of states at the leaves of the tree, are sufficient to determine that parameter. Thus, some parameters of a model can be identifiable while others may not be. For the case of Cavender-Farris and Jukes-Cantor, for example, the unrooted tree topology is identifiable, the substitution probabilities are identifiable, but the location of the root is not.

Statistical Consistency. Statistical consistency is another important concept, but is a property of a method rather than a model. Thus, a phylogeny reconstruction method is a statistically consistent estimator of a model parameter (such as the tree topology) if, for all model trees in that model, the error of the estimated value for the parameter decreases to zero as the number of samples obtained from the model increases.

A simple example of this that is not phylogenetic is estimating the bias in a coin, i.e., the probability that a random coin flip will come up heads. If the coin is *fair*, then this probability is 0.5; however, since the coin may be biased one way or the other, it can be of interest to estimate this probability by doing many random coin flips. Suppose the true probability for a given coin is q, and so we do not require that $q = 0.5$. If we toss the coin many times, and then take the number of times a head shows up and divide by the number of coin tosses, we get an estimate $\hat{q}$ of q. Note that our estimate can have some error, and so $\epsilon = |(\hat{q}) - q| \neq 0$. This estimator is statistically consistent if $\epsilon \to 0$ as the number of tosses increases.

In the context of phylogeny estimation, the parameter of interest could be the underlying unrooted tree topology. To define what we mean by statistical consistency, we need to quantify the error in the estimator. A typical way of quantifying the error in a tree estimator is the Robinson-Foulds distance, which is the number of edges in the estimated tree or the true tree that are not in both trees, where each edge is identified by the bipartition it induces on the leaf set. Thus, the Robinson-Foulds (RF) distance ranges from 0 to $2n - 6$, where n is the number of leaves in the tree. When the RF distance is 0, the two trees are identical topologically.

Of course, not all methods have the guarantee of producing the correct tree with high probability when given a sufficiently large number of characters, but some methods do. Interestingly, maximum parsimony and maximum compatibility do not have this guarantee, nor does UPGMA (the agglomerative clustering method used in lexicostatistics). However, Bayesian methods and maximum likelihood methods do, and so do some polynomial time distance-based methods.

6.2 Calculating the probability of a site pattern

A question we can ask here is how a model tree (for example, a Cavender-Farris model) defines a probability distribution on sequences. Suppose we are given a CF model tree and a single bit (0 or 1) at each leaf. We would like to know how to define the probability of this "site pattern", using the model tree substitution probability parameters.

The simplest (albeit computationally most expensive) way to do this calculation is to compute the probability of each pattern for each way of setting the states at the internal nodes (including the root), and then sum over all those probabilities. For a given assignment of states to internal nodes, the probability of the data is easy to calculate! First the probability of the root having state 0 is 0.5, and similarly the probability of being state 1 is also 0.5. Then on any edge e on which there is a change of state, the probability is $p(e)$, and edges on which there is no change of state the probability is $1 - p(e)$. These probabilities are all multiplied together, since they are independent.

This is a brute-force calculation, and so will take $O(n2^n)$ time, where there are n leaves in the tree (if each multiplication is counted as having unit cost). To do this more efficiently, we can use a simple dynamic programming algorithm, quite similar to the algorithm used for maximum parsimony.

Note that where the tree is rooted has no impact on the result of the calculation, since the model is time-reversible.

6.3 Calculating the probability of a set of sequences

If we assume that all sites evolve independently and identically, then the probability of a set of sequences of length greater than 1 is just the product, over all sites, of the probability of the pattern for that site.

This simple observation allows us to define the likelihood of a model tree as the probability of the data being generated by that model tree.

6.4 Bayesian methods

Bayesian methods, by definition, are based upon explicit parametric models of evolution. They are also "likelihood-based" in that they calculate likelihoods of trees based upon that explicit parametric mathematical model of evolution. They differ from maximum likelihood methods in that they do not attempt to estimate the parameters of the evolutionary model in order to maximize the probability of producing the data. Instead, they perform a "random walk" through model tree space (where the tree and the associated parameters of evolution are provided), by computing the probability of producing the observed character data for each model tree it visits, and then accepting the new model tree if the probability is larger. If the probability is smaller, then the new model tree may be accepted with some non-zero probability (but less than 1). Bayesian methods thus have to operate for a very long time, doing many proposals, until "stationarity" is reached. When the MCMC chain reaches stationarity, a sample of the model trees it visits is then taken, and the summary statistics of that sample are calculated. These give estimates of the support for the various parameters of the evolutionary process. Mostly, however, these summary statistics are used to estimate support for the different branches of the estimated tree(s).

Of fundamental importance, then, is the specific parameterized statistical model underlying the Bayesian method. To date, almost all of the Bayesian methods used in linguistic phylogenetics assume binary characters, based upon "presence/absence" of a feature, and assume that all the binary characters evolve identically and independently (see, for example, the papers by Gray and Atkinson). To make these methods applicable to linguistics, the naturally multi-state character datasets are modified so that they are binary character matrices. As we discussed in Section 4.7, these binary characters behave quite differently than the multi-state characters. For example, the multi-state characters may be homoplasy-free, but the binary characters will not generally be homoplasy-free. Clearly, the binary characters derived from multi-state characters do not evolve independently, nor are they likely to evolve under the same model. Thus, these Bayesian methods are based upon models that are clearly somewhat problematic for linguistic analyses.

6.5 Maximum Likelihood methods

Maximum likelihood methods are similar to Bayesian methods in that they are based upon parametric models of evolution, and compute likelihood scores. They differ in that they also seek to find optimal parameter settings, so that the likelihood is maximized. Thus, maximum likelihood is an optimization problem, while Bayesian estimation is not.

7 Other phylogenetic estimation issues

7.1 Inferring ancestral states

A fundamental problem of great interest is estimating the characteristics of the ancestral taxa.

In linguistics, these ancestral taxa are typically called proto-languages, and various approaches have been used to estimate the properties of these proto-languages. However, standard approaches are not reliable. For example, assuming that if a state is shared by more than half the known languages, it will be shared by the language at the root, is not accurate. Knowing the tree, however, can help with these estimations. For example, in the Indo-European tree, under the assumption that Anatolian is the first child of Proto-Indo-European (PIE), then the only estimations that can be reliably made are for those characters for which some Anatolian has one state and some other non-Anatolian language also has that state. Then that state must also be exhibited at the root, unless the character evolved homoplastically.

7.2 Locating the root

Note that when the characters are directed, it is possible to narrow down the location for the root – sometimes quite precisely. Here we show some examples where this can be done. Suppose the tree is given by $(A, (B, (C, (D, E))))$, and the characters on these languages are given by:

- $A = (0, 0, 0)$
- $B = (0, 1, 0)$
- $C = (0, 0, 1)$
- $D = (1, 0, 1)$
- $E = (1, 0, 1)$

If we assume that 0 is the ancestral state, and 1 the derived state, and there is no homoplasy in these characters, then there are only two edges in the tree which could contain the root.

7.3 Estimating dates at internal nodes

Sometimes researchers are particularly interested in estimating dates at internal nodes of the tree. To do this, the first step involves estimating a tree and its branch lengths, two tasks that are relatively reasonably well addressed. The second step involves combining those branch length estimations with dates at certain nodes of the estimated tree provided by external evidence.

8 Reticulate evolution

8.1 Introduction

Reticulate evolution covers any history which is not purely tree-like. In biology, this is typically from horizontal gene transfer (HGT) or hybrid speciation; in linguistics, this is typically from loan words, or other types of borrowing between languages that come into contact. The specific models of these transmissions differ between these two contexts, and the algorithms for estimating these reticulate histories (represented by phylogenetic networks instead of trees) differ accordingly.

8.2 Reticulation in linguistics

Languages frequently "borrow" lexemes from other languages, and these borrowings can be undetectable under some circumstances. Similarly, other characteristics can be transmitted "horizontally". For these reasons, a tree model is not always the most appropriate model for the evolutionary process.

8.3 Reticulation in biology