Chapter 1

Parsimony methods

Parsimony methods are the easiest ones to explain, and were also among the first methods for inferring phylogenies. The issues that they raise also involve many of the phenomena that we will need to consider. This makes them an appropriate starting point.

The general idea of parsimony methods was given in their first mention in the scientific literature: Edwards and Cavalli-Sforza's (1963) declaration that that evolutionary tree is to be preferred that involves "the minimum nett amount of evolution". We seek that phylogeny on which, when we reconstruct the evolutionary events leading to our data, there are as few events as possible. This raises two issues. First, we must be able to make a reconstruction of events, involving as few events as possible, for any proposed phylogeny. Second, we must be able to search among all possible phylogenies for the one or ones that minimize the number of events.

A simple example

We will illustrate the problem with a small example. Suppose that we have five species, each of which has been scored for 6 characters. In our example, the characters will each have two possible states, which we call 0 and 1. The data are shown in Table 1.1. The events that we will allow are changes from $0 \rightarrow 1$ and from $1 \rightarrow 0$. We will also permit the initial state at the root of a tree to be either state 0 or state 1.

Evaluating a particular tree

To find the most parsimonious tree, we must have a way of calculating how many changes of state are needed on a given tree. Suppose that someone proposes the phylogeny in Figure 1.1. The data set in our example is small enough that we can find by "eyeball" the best reconstruction of evolution for each character. Figures 1.2-1.6 show the best character state reconstructions for characters

Table 1.1: A simple data set with 0/1 characters.

	Characters					
Species	1	2	3	4	5	6
Alpha	1	0	0	1	1	0
Beta	0	0	1	0	0	0
Gamma	1	1	0	0	0	0
Delta	1	1	0	1	1	1
Epsilon	0	0	1	1	1	0

1 through 6. Figure 1.2 shows character 1 reconstructed on this phylogeny. Note that there are two equally good reconstructions, each involving just one change of character state. They differ in which state they assume at the root of the tree, and they also differ in which branch they place the single change. The arrows show the placements of the changes, and the shading shows in which parts of the phylogeny the two states are reconstructed to exist. Figure 1.3 shows the three equally good reconstructions for character 2, which needs two changes of state. Figure

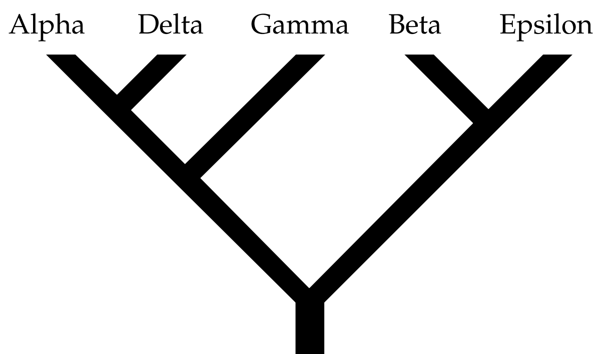

Figure 1.1: A phylogeny which we want to evaluate using parsimony

1.4 shows the single reconstruction for character 3, involving one change of state. Figure 1.5 shows the reconstructions (there are two of them) for character 4. These

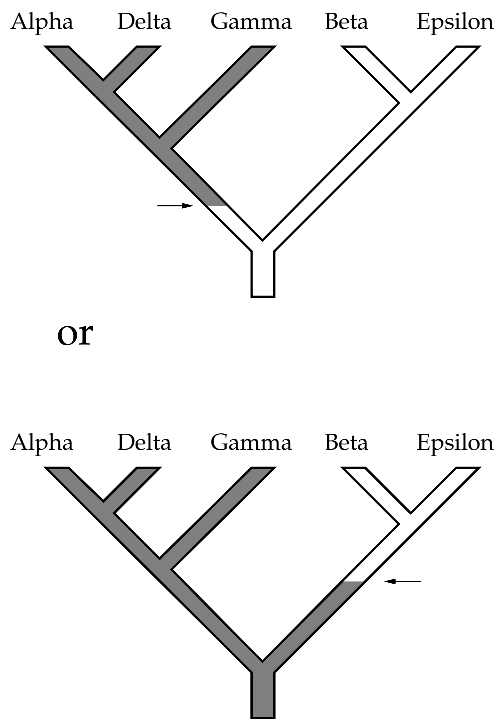

Figure 1.2: Alternative reconstructions of character 1 on the phylogeny of Figure 1.1. The white region of the tree is reconstructed as having state 0, the shaded region as having state 1. The two reconstructions each have one change of state. The changes of state are indicated by arrows.

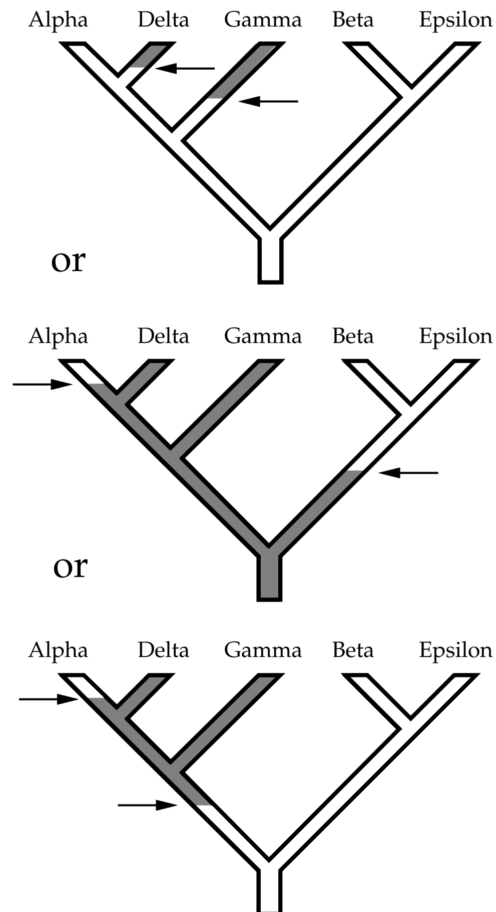

Figure 1.3: Reconstructions of character 2 on the phylogeny of Figure 1.1. The white regions have state 0, the shaded region state 1. The changes of state are indicated by arrows.

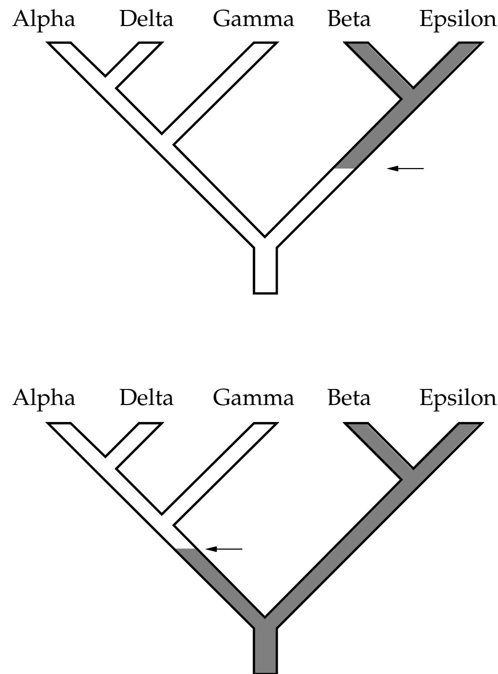

Figure 1.4: Reconstruction of character 3 on the phylogeny of Figure 1.1. The graphical conventions are the same as in the previous figures.

are the same as for character 5, as these two characters have identical patterns.. They require two changes. Finally, Figure 1.6 shows the single reconstruction for character 6. This requires one change of state.

The net result of these reconstructions is that the total number of changes of character state needed on this tree is $1 + 2 + 1 + 2 + 2 + 1 = 9$. Figure 1.7 shows the reconstructions of the changes in state on the tree, making particular arbitrary choices where there is a tie. However, consideration of the character distributions suggests an alternative tree, shown in Figure 1.8, which has one fewer change, needing only 8 changes of state. Considering all possible trees shows that this is the most parsimonious phylogeny for these data. The Figure shows the locations of all of the changes (making, as before, arbitrary choices among alternative reconstructions for some of the characters).

In the most parsimonious tree, there are 8 changes of state. The minimum number we might have hoped to get away with would be 6, as there are 6 charac-

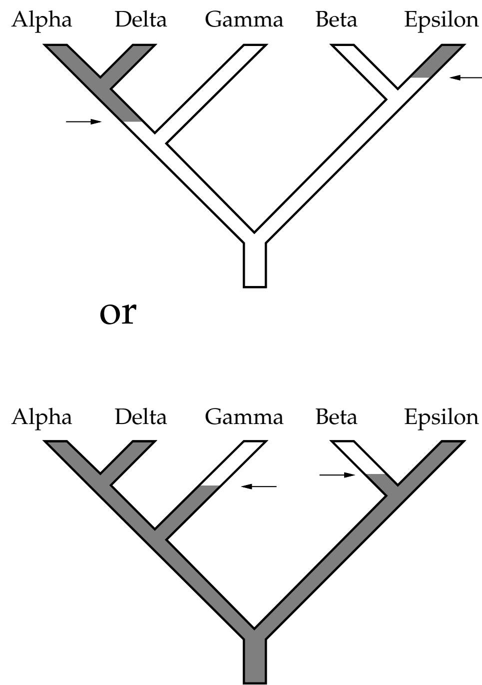

Figure 1.5: Reconstruction of character 4 on the phylogeny of Figure 1.1. This is the same as the reconstruction for character 5 as well. The graphical conventions are the same as in the previous figures.

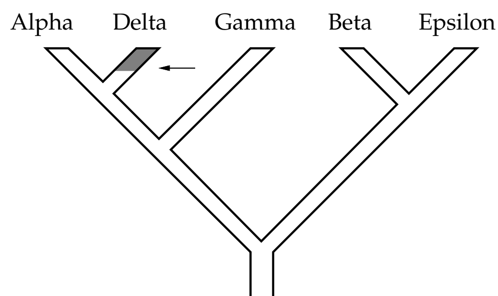

Figure 1.6: Reconstruction of character 6 on the phylogeny of Figure 1.1.

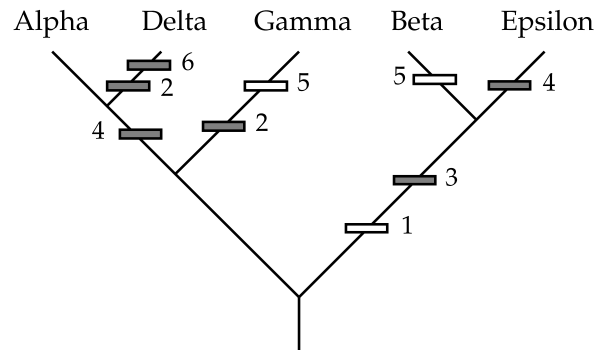

Figure 1.7: Reconstruction of all character changes on the phylogeny of Figure 1.1. The changes are shown as bars across the branches, with a number next to each indicating which character is changing. The shading of each box indicates which state is derived from that change.

ters, each of which has two states present in the data. Thus we have two "extra" changes. Having some states arise more than once on the tree is called *homoplasy*.

Rootedness and unrootedness

Figure 1.9 shows another tree. It also requires 8 changes, as shown in that figure. In fact, these two most parsimonious trees are the same in one important respect – they are both the same tree when the roots of the trees are removed. Figure 1.10 shows that unrooted tree. The locations of the changes are still shown (and still involve some arbitrary choices), but they are no longer shaded in to show the direction of the changes. There are many rooted trees, one for each branch of the unrooted tree in Figure 1.10, and all have the same number of changes of state. In fact, the number of changes of state will depend only on the unrooted tree, and not at all on where the tree is then rooted. This is true for the simple model of character change that we are using ($0 \rightleftharpoons 1$). It is also true for any model of character change that has one simple property: that if we can go in one change from state a to state b, we can also go in one change from state b to state a.

When looking at the alternative placements of changes of state, it actually matters whether one is looking at a rooted or an unrooted tree. In Figure 1.3, there are three different reconstructions. The last two of them differ only by whether a single change is placed to the left or to the right of the root. Once the tree is unrooted, these last two possibilities become identical. So the rooted tree has three possible reconstructions of the changes of this state, but the unrooted tree has only two.

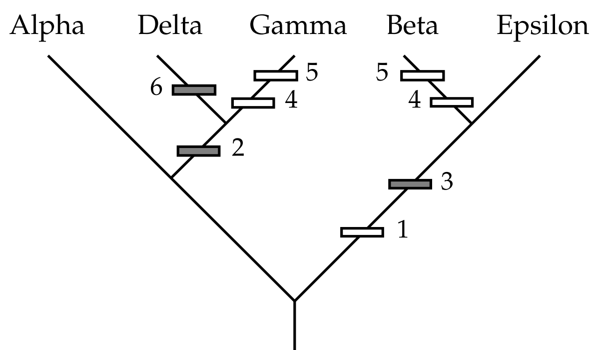

Figure 1.8: Reconstruction of all changes on the most parsimonious phylogeny for the data of Table 1.1. It requires only 8 changes of state. The changes are shown as bars across the branches, with a number next to each indicating which character is changing. The shading of each box indicates which state is derived from that change.

Methods of rooting the tree

Biologists want to think of trees as rooted, and thus have been interested in methods of placing the root in an otherwise unrooted tree. There are two methods: the outgroup criterion and use of a molecular clock. The outgroup criterion amounts to knowing the answer in advance. Suppose that we have a number of great apes, plus a single old world (cercopithecoid) monkey. Suppose that we know that the great apes are a monophyletic group. If we infer a tree of these species, we then know that the root must be on the lineage that connects the cercopithecoid monkey to the others. Any other placement would make the apes fail to be monophyletic, because there would then be a lineage leading away from the root with a subtree that included the cercopithecine and also some, but not all, of the apes. We place the root outside of the ingroup, so that it is monophyletic.

The alternative method is to make use of a presumed clocklike behavior of character change. In molecular terms, this is the "molecular clock". If an equal amount of change were observed on all lineages, there should be a point on the tree which has equal branch lengths from there to all tips. With a molecular clock, it is only the expected amounts of change that are equal; the observed amounts may not be. We hope to find a root that makes the amounts of change approximately equal on all lineages. In some methods, we constrain the tree to remain clocklike by making sure that no tree is inferred that violates this constraint. If instead we infer a tree without maintaining this constraint, we can try to remedy this by

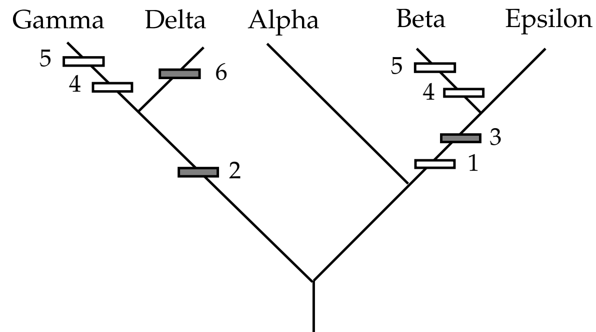

Figure 1.9: Another rooted tree with the same number of changes of state.

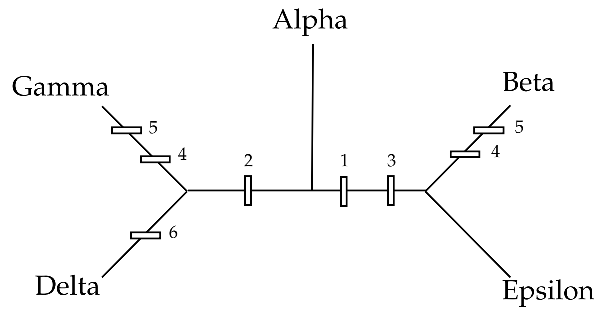

Figure 1.10: The unrooted tree corresponding to Figures 1.8 and 1.9.

finding, after the fact, a point on the tree approximately equidistant from the tips. This may be difficult.

Branch lengths

Having found an unrooted tree, we might want to locate the changes on it, and find out how many occur in each of the branches. We have already seen that there can be ambiguity as to where the changes are. That in turn means that we cannot necessarily count the number of changes in each branch. One possible alternative is to average over all possible reconstructions of each character for which there is ambiguity in the unrooted tree. This has the advantage that, although this can leave fractional numbers of changes in some branches, at least they must add up to the total number of changes in the tree. Figure 1.11 shows the same tree as Figure 1.7 (not the most parsimonious tree), using these branch lengths. The lengths of the branches are shown visually, and also given as numbers beside each branch.

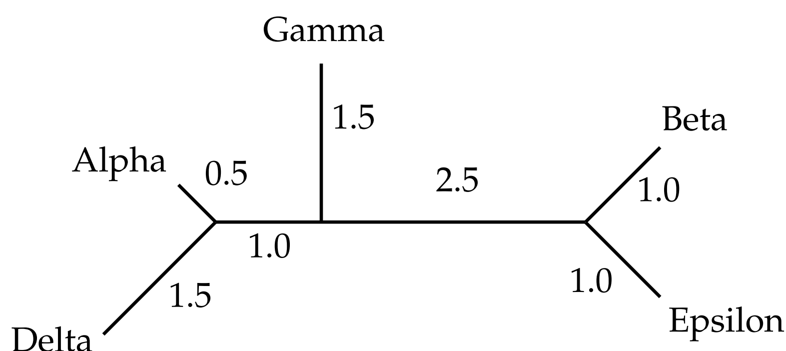

Figure 1.11: The tree of Figure 1.1, shown as an unrooted tree with branch lengths computed by averaging all equally parsimonious reconstructions.

Unresolved questions

Although we have mentioned many of the issues involved in using parsimony, we have not actually given the algorithms for any of them. In every case we simply reconstructed character states by "eyeball", and similarly we searched the set of possible trees by informal means. Among the issues that need to be discussed are:

- Particularly for larger data sets, we need to know how to count the number of changes of state by use of an algorithm.

- We need to know the algorithm for reconstructing states at interior nodes of the tree.
- We need to know how to search among all possible trees for the most parsimonious ones, and how to infer branch lengths.
- All of the discussion here has been for a simple model of 0/1 characters. What do we do with DNA sequences, that have 4 states, or with protein sequences, that have 20? How do we handle more complex morphological characters?
- There is the crucial issue of justification. Is it reasonable to use the parsimony criterion? If so, what does it implicitly assume about the biology?
- Finally, what is the statistical status of finding the most parsimonious tree? Is there some way we can make statements about how well-supported a most parsimonious tree is over the others?

Much work has been done on these questions, and it is this that we cover in the next few chapters.

Chapter 2

Counting evolutionary changes

Counting the number of changes of state on a given phylogeny requires us to have some algorithm. The first such algorithms for discrete-states data were given by Camin and Sokal (1965), for a model with unidirectional changes, and by Kluge and Farris (1969) and Farris (1970) for bidirectional changes on a linear ordering of states. We will discuss here two algorithms that generalize these, one by Fitch (1971) and the other by Sankoff (1975) and Sankoff and Rousseau (1975). Both have the same general structure. We evaluate a phylogeny character by character. For each character, we consider it as a rooted tree, placing the root wherever seems appropriate. We update some information down a tree; when we reach the bottom the number of changes of state is available. In both cases, the algorithm does *not* function by actually locating changes or by actually reconstructing interior states at the nodes of the tree. Both are examples of the class known as *dynamic programming* algorithms.

In the previous chapter we found the most parsimonious assignments of ancestral states, and did so by "eyeball". In the present chapter we show how the counting of changes of state can be done more mechanically.

The Fitch algorithm

The Fitch (1971) algorithm was intended to count the number of changes in a bifurcating tree with nucleotide sequence data, where any one of the four bases {A, C, G, T} can change to any other. It also works generally for any number of states, provided one can change from any one to any other. This multistate parsimony model was named *Wagner parsimony* by Kluge and Farris (1969). Fitch's algorithm thus works perfectly for the $0 \rightleftharpoons 1$ case as well. (In fact, Farris (1970) gave a version of this algorithm, for the special case of a linear series of discrete states.) The algorithm at first seems to be mumbo-jumbo. It is only after understanding how the Sankoff algorithm works that one can see why it works, and that it is an algorithm of the same general class. We will explain the Fitch algorithm by

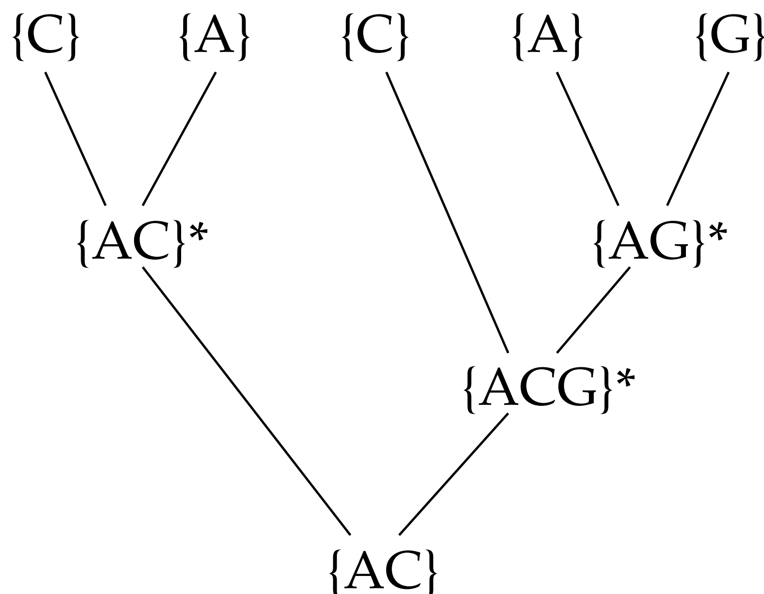

Figure 2.1: An example of the Fitch algorithm applied to a single site. The sets generated at each node are shown.

use of an example, which is shown in Figure 2.1. The Fitch algorithm considers the sites (or characters) one at a time. At each tip in the tree, we set up a set containing those nucleotides (states) that are observed, or are compatible with the observation. Thus if we see an A, we create the set $\{A\}$. If we see an ambiguity such as R (purine), we create the set $\{AG\}$. Now we move down the tree. In algorithmic terms, we do a postorder tree traversal. At each interior node we create a set which is the intersection of sets at the two descendant nodes. However, if that set is empty, we instead create the set that is the union of the two sets at the descendant nodes. Every time we create such a union, we also count one change of state.

In Figure 2.1, we are evaluating a tree with five species. At the particular site, we have observed the bases C, A, C, A, and G in the five species, where we give them in the order in which they appear in the tree, left to right. For the left two, at the node which is their immediate common ancestor we first attempt to construct the intersection of the two sets. But as $\{C\} \cap \{A\} = \emptyset$, we instead construct the union $\{C\} \cup \{A\} = \{AC\}$ and count one change of state. Likewise, for the right-most pair of species, their common ancestor will be assigned state $\{AG\}$, since

$\{A\} \cap \{G\} = \emptyset$, and we count another change of state. The node below it now can be processed. $\{C\} \cap \{AG\} = \emptyset$, so we construct the union $\{C\} \cup \{AG\} = \{ACG\}$ and count a third change of state. The node at the bottom of the tree can now be processed. $\{AC\} \cap \{ACG\} = \{AC\}$, so we put {AC} at that node. We have now counted 3 changes of state. A moment's glance at the figure will verify that 3 is the correct count of the number of changes of state. On larger trees the moment's glance will not work, but the Fitch algorithm will continue to work.

The Fitch algorithm can be carried out in a number of operations which is proportional to the number of species (tips) on the tree. One might think that one would also need to multiply this by the number of sites, since we are computing the total number of changes of state over all sites. But we can do better than that. Any site which is invariant, which has the same base in all species (such as $AAAAA$), will never need any changes of state and can be dropped from the analysis without affecting the number of changes of state. Other sites, that have a single variant base present in only a single species (such as, reading across the species, $ATAAA$), will require a single change of state on all trees, no matter what their structure. These too can be dropped, though we may want to note that they will always generate one more change of state each. In addition, if we see a site that has the same pattern (say $CACAG$) that we have already seen, we need not recompute the number of changes of state for that site, but can simply use the previous result. Finally, the symmetry of the model of state change means that if we see a pattern, such as $TCTCA$, that can be converted into one of the preceding patterns by changing the four symbols, it too does not need to have the number of changes of state computed. Both $CACAG$ and $TCTCA$ are patterns of the form $xyxyz$, and thus both will require at least 2 changes of state. Thus the effort rises slower than linearly with the numbers of sites, in a way that is dependent on how the data set arose.

One might think that we could use the sets in Figure 2.1 to reconstruct ancestral states at the interior nodes of the tree. They certainly can be used in that process, but they are not themselves reconstructions of the possible nucleotides, nor do they even contain the possible nucleotides that a parsimony method would construct. For example, in the common ancestor of the rightmost pair of species, the set that we construct is $\{AG\}$. But a careful consideration will show that if we put C at all interior nodes, including that one, we attain the minimum number of changes, 3. But C is not a member of the set that we constructed. At the immediate ancestor of that node, we constructed the set $\{ACG\}$. But of those nucleotides, only A or G are possible in assignments of states to ancestors that achieve a parsimonious result.

Sankoff's algorithm

The Fitch algorithm is enormously effective, but it gives us no hint as to why it works, nor does it show us what to do if we want to count different kinds of changes differently. The Sankoff algorithm is more complex, but its structure is

more apparent. It starts by assuming that we have a table of the cost of changes between each character state and each other state. Let's denote by c_{ij} the cost of change from state i to state j. As before, we compute the total cost of the most parsimonious combinations of events by computing it for each character. For a given character, we compute, for each node k in the tree, a quantity $S_k(i)$. This is interpreted as the minimal cost, given that node k is assigned state i, of all the events upwards from node k in the tree. In other words, the minimal cost of events in the subtree which starts at node k and consists of everything above that point.

It should be immediately apparent that if we can compute this for all nodes, we can compute it for the bottom node in the tree, in particular. If we can compute it for the bottom node (call that node 0), then we can simply choose the minimum of these values:

$$S = \min_i S_0(i) \tag{2.1}$$

and that will be the total cost we seek, the minimum cost of evolution for this character.

At the tips of the tree the $S(i)$ are easy to compute. The cost is 0 if the observed state is state i, and infinite otherwise. If we have observed an ambiguous state, the cost is 0 for all states that it could be, and infinite for all the rest. Now all we need is an algorithm to calculate the $S(i)$ for the immediate common ancestor of two nodes. This is very easy to do. Suppose that the two descendant nodes are called l and r (for "left" and "right"). For their immediate common ancestor, node a, we need only compute

$$S_a(i) = \min_j \left[c_{ij} + S_l(j)\right] + \min_k \left[c_{ik} + S_r(k)\right] \tag{2.2}$$

The interpretation of this equation is immediate. The smallest possible cost given that node a is in state i is the cost c_{ij} of going from state i to state j in the left descendant lineage, plus the cost $S_l(j)$ of events further up in that subtree given that node l is in state j. We select that value of j which minimizes that sum. We do the same calculation in the right descendant lineage, which gives us the second term of (2.2). The sum of these two minima is the smallest possible cost for the subtree above node a, given that node a is in state i.

This equation is applied successively to each node in the tree, working downwards (doing a postorder tree traversal). Finally it computes all the $S_0(i)$, and then (2.1) is used to find the minimum cost for the whole tree.

The process is best understood by an example, the example that we already used for the Fitch algorithm. Suppose that we wish to compute the smallest total cost for the given tree, where we weight transitions (changes between two purines or two pyrimidines) 1, and weight transversion (changes between a purine and a pyrimidine or between a pyrimidine and a purine) 2.5. Figure 2.2 shows the cost matrix and the tree, with the $S(i)$ arrays at each node. You can verify that these are correctly computed. For the leftmost pair of tips, for example, we observe states C and A, so the S arrays are respectively $(\infty, 0, \infty, \infty)$ and $(0, \infty, \infty, \infty)$. Their

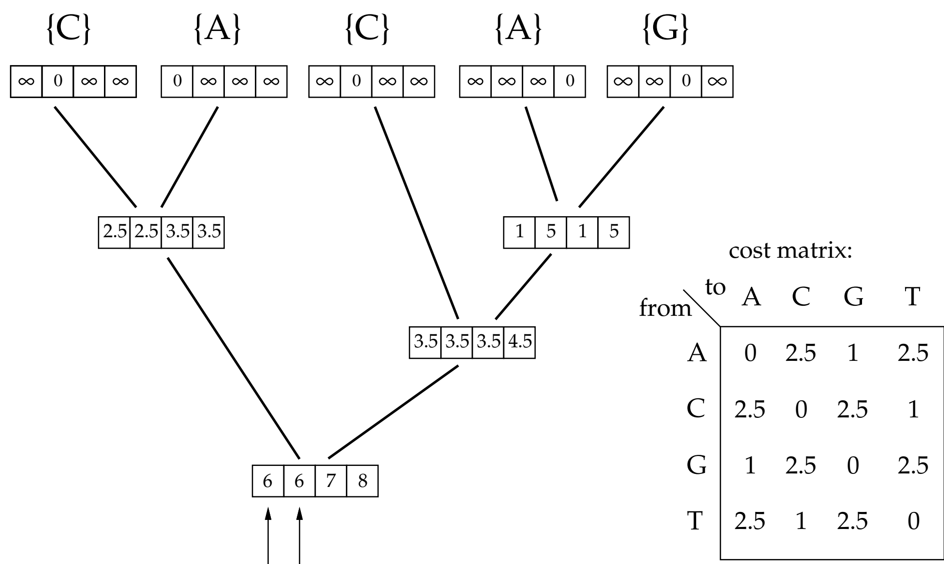

Figure 2.2: The Sankoff algorithm applied to the tree and site of the previous figure. The cost matrix used is shown, as well as the S arrays computed at each node of the tree.

ancestor has array $(2.5, 2.5, 3.5, 3.5)$. This is because if the ancestor has state A, the least cost is 2.5, for a change to a C on the left lineage and no change on the right. If it has state C, the cost is also 2.5, for no change on the left lineage combined with change to an A on the right lineage. For state G the cost is 3.5, because we can at best change to C on the left lineage (at cost 2.5) and to state A on the right lineage, for a cost of 1. Similarly for T, where the costs are 1 + 2.5 = 3.5.

Another node where the result may be less obvious is the common ancestor of the rightmost three species, where the result is $(3.5, 3.5, 3.5, 4.5)$. The first entry is 3.5 because you could have changed to C on the left branch (2.5 changes plus 0 above that) and had no change on the right branch (0 changes plus 1 above that). That totals to 3.5; no other scenario achieves a smaller total. The second entry is 3.5 because you could have had no change on the left branch (0+0) and a change to A or to G on the right one (each 2.5 + 1). The third entry is 3.5 for much the same reason the first one was. The fourth entry is 4.5 because it could have changed on the left branch from T to C (1 + 0), and on the right branch from T to A or T to G (2.5 + 1) and these total to 2.5.

Working down the tree, we arrive at the array $(6, 6, 7, 8)$ at the bottom of the tree. The minimum of these is 6, which is the minimum total cost of the tree for

this site. When the analogous operations are done at all sites, and their minimal costs added up, the result is the minimal cost for evolution of the dataset on the tree.

The Sankoff algorithm is a dynamic programming algorithm, because it solves the problem of finding the minimum cost by first solving some smaller problems and then constructing the solution to the larger problem out of these, in such a way that it can be proven that the solution to the larger problem is correct. An example of a dynamic programming algorithm is the well-known least-cost-path-through-a-graph algorithm. We will not describe it in detail here, but it involves gradually working out the costs of paths to other points in the graph, working outwards from the source. It makes use of the costs of paths to points to work out the costs of paths to their immediate neighbors, until we ultimately know the lengths of the lowest-cost paths from the source to all points in the graph. This does not involve working out all possible paths, and it is guaranteed to give the correct answer.

An attempt to simplify computations by Nixon and Wheeler (1990) has been shown by Swofford and Siddall (1997) to be incorrect.

Connection between the two algorithms

The Fitch algorithm is a close cousin of the Sankoff algorithm. Suppose that we made up a variant of the Sankoff algorithm in which we keep track of an array of (in the nucleotide case) four numbers, but associated them with the bottom end of a branch instead of the node at the top end of a branch. We could then develop a rule similar to (2.2) that would update this array down the tree. For the simple cost matrix that underlies the Fitch algorithm, it will turn out that the numbers in that array are always either x or $x + 1$. This is true because one can always get from any state to any other with penalty 1. So you can never have a penalty that is more than one greater than the minimum that is possible at that point on the tree. Fitch's sets are simply the sets of nucleotides that have the minimum value x rather than the higher value of $x + 1$. A careful consideration of the updating rule in Sankoff's algorithm in this case will show that it corresponds closely to the set operations that Fitch specified. Because it is updating the quantities at the bottom end rather than at the top end of each branch, the Fitch algorithm is not a special case of the Sankoff algorithm.

Using the algorithms when modifying trees

Views

For most of the parsimony methods that we will discuss, the score of a tree is unaltered when we reroot the tree. We can consider any place in the tree as if it were the root. Looking outward from any branch, we see two subtrees, one at each end of the branch. Taking the root to be on the branch, we can use the Fitch or Sankoff parsimony algorithms to move "down" the tree towards that point,

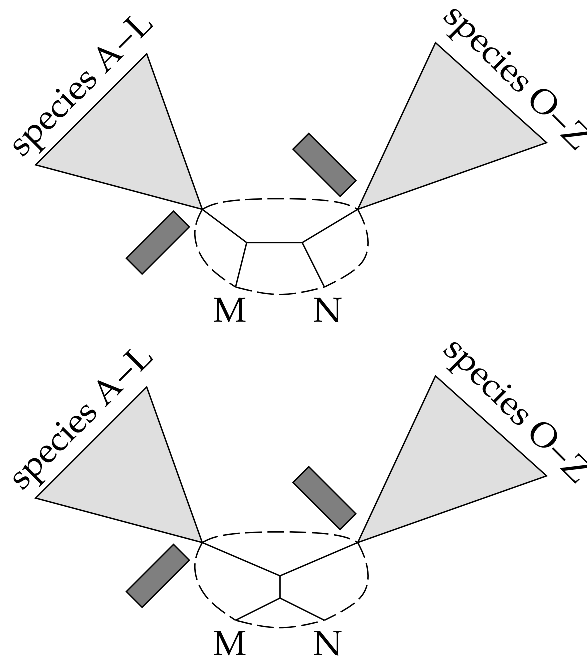

Figure 2.3: Two trees illustrating the use of the conditional scores of the Fitch and Sankoff methods in economizing on computations when rearranging a tree. The two gray rectangles stand for the views for a character in the two subtrees. When species M and N are involved in a rearrangement, the views can be used as if they summarized the data at a tip. They remain unaltered when M and N are rearranged, and the rearrangement can be evaluated by doing calculations entirely within the region outlines by the dashed curve.

calculating the arrays of scores for a character. There will be arrays at the two ends of our branch. This can be thought of as "views" summarizing the parsimony scores in these two subtrees, for the character. Each interior node of the tree will have three (or more) views associated with it: one for each branch that connects to that node. Thus in the tree in Figure 2.2 we see one view for the node above and to the right of the root. It shows the view up into the subtree that has the three rightmost species. But there are two other views that we could have calculated as well. One would show the view looking down at that node from the center species, and one the view looking down at that node from the branch that leads

to the two rightmost species. If the node had had four branches connecting to it, there would have been four views possible.

It is worth noting that views also exist for likelihood methods and for some algorithms for distance matrix methods.

Using views when a tree is altered

Both the Fitch and Sankoff algorithms use such views, though they only compute one view at each internal node, the one that looks up at it from below. We can calculate views anywhere in the tree, by passing inwards toward that point from tips. This can be convenient when rearranging or otherwise altering trees. Figure 2.3 shows an example. The two gray rectangles are the views for a character for the two subtrees (which are the large triangles). When we rearrange the two species M and N locally, without disrupting the structure of either subtree, we can compute the parsimony score for the whole tree by using the numbers in the rectangles, and doing all of our computations within the regions enclosed by the dashed curves. This enables a fast diagnosis of local rearrangements.

This method of economizing on the effort of computing parsimony scores was first described in print by Gladstein (1997). His discussion codifies methods long in use in the faster parsimony programs, but not previously described in print.

When we come to discuss likelihood methods later in the book, we will see views that play a very similar role. They allow similar economies, but are limited by the fact that as one branch length is changed, others elsewhere in the tree must also be altered for the tree to be optimal. In some least squares algorithms for distance matrix methods, there are conditional quantities that behave similarly.

Further economies

There are some additional economies, beyond Gladstein's method, that help speed up parsimony calculations. Ronquist (1998a) points out an economy that can have a large effect when we use a Fitch or Sankoff algorithm and compute views at all nodes, looking in all directions. We have been discussing the tree as if it were rooted, but in most cases it effectively be an unrooted tree.

When a tree is modified in one part, all the inward-looking views may need updating (all those that summarize subtrees that include the modified region). Ronquist points out that we do not need to go through the entire rest of the tree modifying these views. As we work outward from the modified region, if we come to a view that looks back in, and that ends up not being changed when it is reconsidered, we need go no further in that direction, as all further views looking back in that way will also be unchanged. This can save a considerable amount of time. We shall see other savings when we discuss tree rearrangement below.