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 that 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. The sets 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 C are possible in assignments of states to ancestors that achieve a parsimonious result.

The Sankoff 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 these values for all nodes, we can compute them for the bottom node in the tree, in particular. If

we can compute them 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

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

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 the value of j that minimizes that sum. We do the same calculation in the right descendant lineage, which gives us the second term of equation 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 ancestor has array $(2.5, 2.5, 3.5, 3.5)$. The reasoning is: 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. We can reason similarly for T, where the costs are 1 + 2.5 = 3.5.

The result may be less obvious at another node, 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

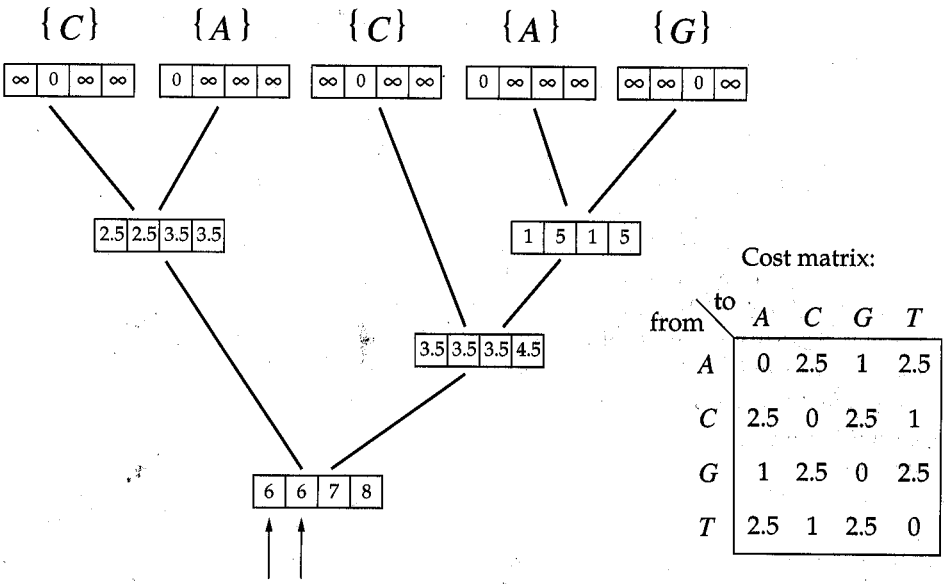

Figure 2.2: The Sankoff algorithm applied to the tree and site of the previous figure. The cost matrix used is shown, as well as the S arrays computed at each node of the tree.

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 4.5.

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 data set 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 Wheeler and Nixon (1994) 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 equation 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, 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 could show the view looking down at that node from the center species, and the other the view looking down at that node from the branch that leads to the