Parsimony: Counting Changes

Fredrik Ronquist and Peter Beerli

August 31, 2005; August 29, 2007

1 Counting Evolutionary Changes

Reference: Felsenstein Chapters 1 and 2 (pp. 1-18), Chapter 6 (pp. 67-72), Chapter 7 (pp. 73-86)

Parsimony methods are perhaps the simplest methods used for inferring evolutionary trees, so we will start with them. Parsimony means "economy in the use of means to an end; especially: economy of explanation in conformity with Occam's razor" [Merriam-Webster Online]. In the context of evolutionary inference, the idea is to minimize the amount of evolutionary change. Another way of understanding parsimony methods is that they are cost minimization procedures, where the cost is a measure of the amount of evolutionary change. In the typical case, we are dealing with discrete characters, and the natural measure of evolutionary change is simply the number of changes between states. However, there are many other possibilities as well, as we will discover.

Throughout this lecture, we will be assuming that the tree is fixed. We will then use parsimony methods to count the number of changes and to infer the ancestral states in the tree. In principle, the step is small from this to comparing and selecting among trees according to the amount of evolutionary change they imply. However, several practical issues arise when we start searching for trees so we will defer treatment of this topic to the next lecture.

1.1 Fitch Parsimony

To start with, let us assume that we are interested in the evolution of a DNA character, which has four discrete states: A, C, G, and T. In the simplest possible model, we assume that all changes

between all states are allowed and count equally (Fig. 1). According to Felsenstein (2004), this model was initially proposed by Kluge and Farris (1969) and named Wagner parsimony by them but it is more widely known as *Fitch Parsimony* because the first algorithms implementing the model were published by Walter Fitch (1970, 1971). The naming issue is utterly confusing because there is another parsimony model that we will examine shortly that is widely known as Wagner parsimony although Felsenstein calls it "parsimony on an ordinal scale".

Figure 1: Fitch parsimony model for DNA characters. All state changes are possible and count equally. All changes are equally likely.

In simple cases, when there are few state changes in a small tree, it is straightforward to find the number of required changes and the most parsimonious states of the interior nodes in the tree (and hence the position of the changes) (Fig. 2a). When there are more changes, it is quite common for there to be ambiguity both concerning the interior node states and the position of the changes, even though we can still find the minimum number of changes relatively easily in many cases (Fig. 2b).

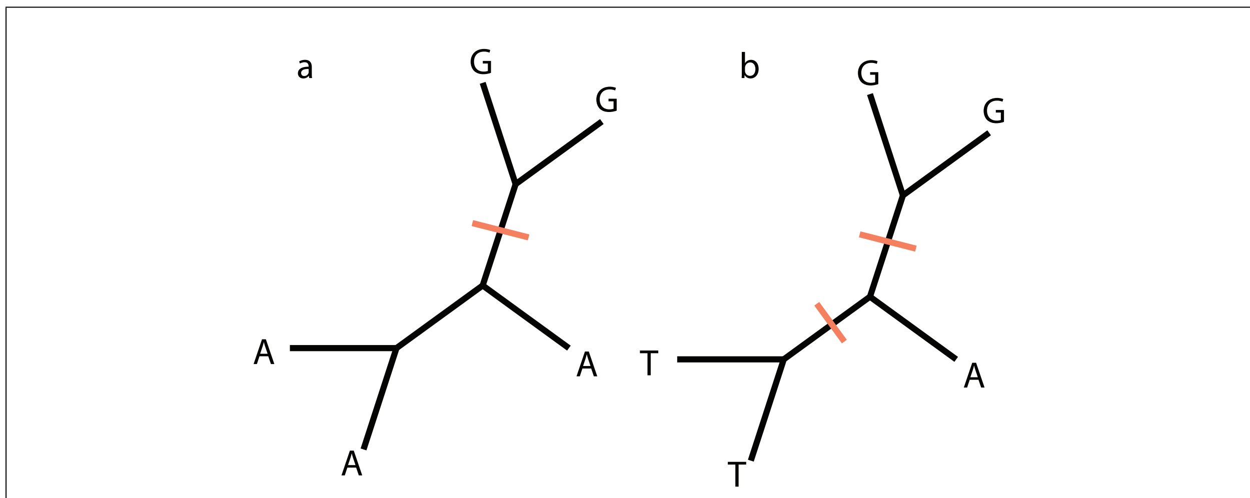

Figure 2: In simple cases, the minimum number of changes and the optimal states at the interior nodes of the tree under Fitch parsimony are obvious. (a) one unambiguous change, (b) an example of a change, at least two others positions of the changes are possible, but the number of minimal changes is 2.

For more complicated trees and larger problems, it is necessary to use an algorithm. The basic idea is to pass through the tree from the tips to the root (a downpass or postorder traversal). At each interior node, we count the minimum number of steps required by the subtree rooted at that node, and we record the ancestral states giving us that number of steps. When we have reached the root node, we know the minimum number of steps for the entire tree. Since it is a recursive algorithm, we only need to specify it for one node p and its two children q and r. Assume that S_i is the set of

observed or inferred minimal states at node i. For instance, if p is a terminal and we have observed an A for it, we have $S_p = \{A\}$. Similarly, if there were ambiguity about the states for a terminal or interior node p, such that the state could be either A or G, then we have $S_p = \{A, G\}$. Now we can calculate the length (or cost) l of a tree using the *Fitch downpass algorithm* by finding sections and unions of state sets (Algorithm 1).

Algorithm 1 Fitch downpass algorithm

$S_p \leftarrow S_q \cap S_r$
if $S_p = \emptyset$ **then**
  $S_p \leftarrow S_q \cup S_r$
  $l \leftarrow l + 1$
end if

The algorithm is relatively easy to understand. If the descendant state sets S_q and S_r overlap, then we do not have to add any steps at this level and the state set of node p will include the states present in both descendants (the intersection of S_q and S_r). If the descendant state sets do not overlap, then we have to add one step at this level and the state set of p will include all states that are present in either one of the descendants (the union of S_q and S_r). States that are absent from both descendants will never be present in the state set of p because they would add two steps to the length of the subtree rooted at p.

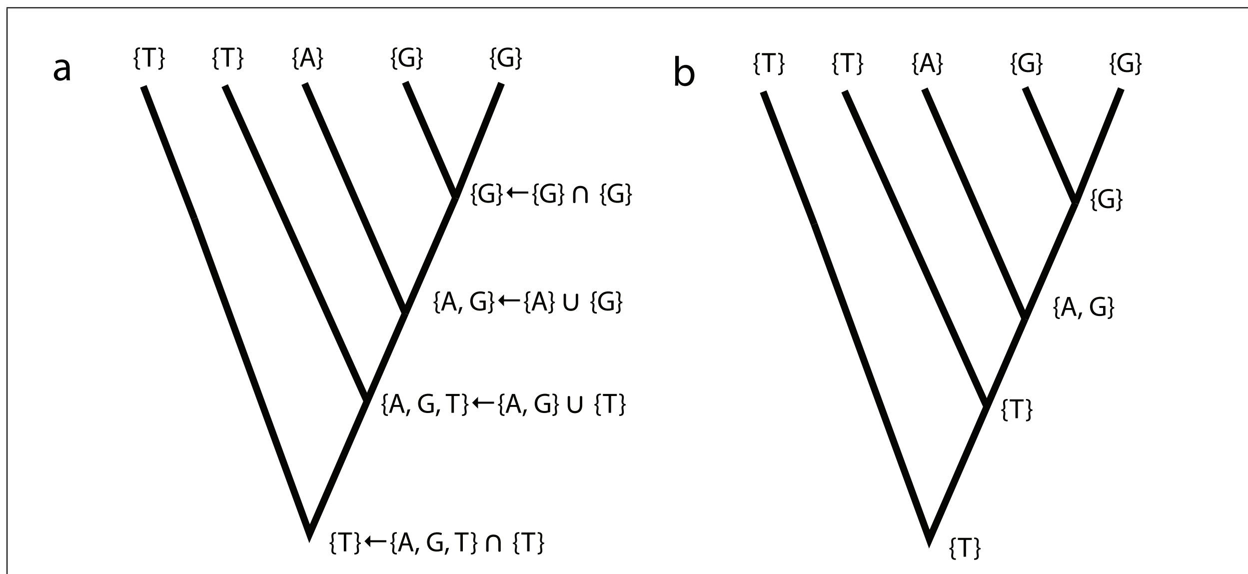

Figure 3: An example of state sets calculated during a Fitch (a) downpass and final ancestral states after the (b) uppass

An example illustrates the procedure clearly (Fig. 3; see also Felsenstein, Fig. 2.1). An interesting aspect of the algorithm is that it is not guaranteed to give you the optimal states at the interior

nodes of the tree. In Felsenstein's example, for instance, the downpass gives you the state set $\{A, G\}$ for one of the nodes but the optimal state set (also known as the *most parsimonious reconstruction (MPR)* for that node) is actually $\{A, C, G\}$. To get the optimal state sets of all the interior nodes in the tree, you need to go through the *Fitch uppass algorithm*.

Assume that we have the final state set F_a of node a, which is the immediate ancestor of node p. We also have the downpass sets of p and its two children q and r; they are labeled S_p, S_q and S_r, respectively, as in the description of the downpass algorithm. Now Fitch's uppass (also known as the final pass) algorithm is based on combining information from these state sets (Algorithm 2). The algorithm is started by noting that, for the root, the final state set is the same as the downpass state set.

Algorithm 2 Fitch uppass algorithm

```
F_p ← S_p ∩ F_a
if F_p ≠ F_a then
  if S_q ∩ S_r ≠ ∅ then
    F_p ← ((S_q ∪ S_r) ∩ F_a) ∪ S_p
  else
    F_p ← S_p ∪ F_a
  end if
end if
```

Fitch's uppass algorithm is much less intuitive than the downpass algorithm but we will nevertheless make an attempt to explain why it works. In the first step, you test whether the downpass state set of p includes all of the states in the final set of a. If it does, then it means that each optimal assignment of final state to a can be combined with the same state at p to give zero changes on the branch between a and p and the minimal number of changes in the subtree rooted at p. Thus, the final set of p must be the same as the set at a because any other state would add steps to the tree length (including any states that might be present in the downpass set of p but not in the final set of a).

If the final set of a includes states that are not present in the downpass set of p, then there must be optimal solutions that have one state at a and a different state at p. This means that there is a potential change on the branch between a and p and we need to see if that change can be pushed onto one of the descendant branches of p without increasing the total tree length. If so, we may have to add states from the final set of a to the downpass set of p. The second part of the algorithm (after the first *if* statement) takes care of this potential addition of states. If the downpass set of p was formed by taking an intersection, then we need to add the states that are in the final set of a

and in the downpass set of either one of the two descendants of p because each of these additional state assignments to p will push a change on the branch between p and a to one of the descendant branches of p without increasing total tree length. If the downpass set of p was formed by a union, then we simply add the states in the final set of a to those in the downpass set of p.

We have described the Fitch algorithms for rooted binary trees. Unrooted trees and polytomous trees require minimal modifications that should be straightforward; we leave those as an exercise. Note that the Fitch algorithms accommodate uncertainty about the state of a terminal in a natural way: we simply express the uncertainty by assigning the terminal a state set containing all the possible states.

1.2 Wagner Parsimony

In some cases, it is natural to assume that the states we are considering form a linear series, usually known in evolutionary biology, particularly systematics, as a *transformation series*. Often cited examples include morphological characters with three states, one of which is intermediate between the two others. Think about an insect antenna with 10, 11 or 12 articles; a process that can be short, intermediate, or long; pubescence that can be sparse, intermediate or dense; or coloration that can be white, gray or black. In such cases, it seems reasonable to model evolution as going from one extreme to the other through the intermediate state. In other words, we allow changes only between the end states and the intermediate state, but not directly between the end states themselves (Fig. 4). Another way of expressing the same thing is that we count a change between the two extreme states as costing twice as much as a change between an extreme state and the intermediate state. Characters for which we would like to use this parsimony model are often referred to as *ordered* or *additive* characters, in contrast to the unordered or non-additive characters for which we use the Fitch model.

Figure 4: The Wagner parsimony model. Going from any of the extreme states to the intermediate state counts as one change, whereas a transition from one extreme state to the other counts as two changes since we are forced to go through the intermediate state.

The Wagner parsimony model can have a large number of states and it can be used to represent quantitative characters either on an ordinal scale or on an interval scale. In the former case, we would simply group the measurements into n ordered groups that would comprise the states of the Wagner parsimony character. In the latter case, we divide the range between the minimum

and maximum value into n equal-length intervals to give us a Wagner parsimony character with n states. In the extreme case, we simply use the raw measurements up to some fixed precision.

Felsenstein (2004, p. ....) describes the Wagner parsimony algorithms using raw measurements, but let us use a slightly modified set description instead. Assume that the state set of a node p is a set of continuous elements $S = \{x, x+1, x+2, ..., y\}$ where $\min(S) = x$ and $\max(S) = y$ (we can also call this set an interval). Now define the operation $S_i \sqcap S_j$ as producing the set of continuous elements from $\{\max(\min(S_i), \min(S_j))\}$ to $\{\min(\max(S_i), \max(S_j))\}$. If S_i and S_j overlap, then this operation simply produces their intersection, but if they do not overlap, the result is a minimum spanning interval connecting the two sets. For instance, $\{2,3,4\} \sqcap \{6,7,8\} = \{4,5,6\}$. As usual, the *Wagner downpass algorithm* finds the downpass interval for a node p from the downpass intervals of its two daughters q and r, and it adds to a global tree length variable l when we need to connect two non-overlapping state sets (Algorithm 3).

Algorithm 3 Wagner downpass algorithm

$S_p \leftarrow S_q \cap S_r$
if $S_p = \emptyset$ **then**
  $S_p \leftarrow S_q \sqcap S_r$
  $l \leftarrow l + (|S_p| - 1)$
end if

The similarities to the Fitch downpass algorithm are obvious. The uppass algorithm is also similar to its Fitch parsimony analog, but it is a little less complex because of the additive nature of the state transition costs. Before describing it, let us define the operation $S_i \sqcup S_j$ as producing the set of continuous elements from $\{\min(\min(S_i), \min(S_j))\}$ to $\{\max(\max(S_i), \max(S_j))\}$. If the two intervals overlap, the result is simply their union, but if they are disjoint then the operation will produce an interval including all the values from the smallest to the largest. For example, $\{3,4\} \sqcup \{6,7\} = \{3,4,5,6,7\}$. With additional notation as above, the *Wagner uppass algorithm* is now easy to formulate (Algorithm 4). The algorithm is slightly simpler than the Fitch uppass but can be justified (proven to be correct) with argumentation similar to that used above for the latter.

Algorithm 4 Wagner uppass algorithm

$F_p \leftarrow S_p \cap F_a$
if $F_p \neq F_a$ **then**
  $F_p \leftarrow ((S_q \sqcup S_r) \cap F_a) \cup S_p$
end if

1.3 Other variants of parsimony

Many other variants of parsimony that count changes slightly differently have been proposed. Some examples include Camin-Sokal parsimony, where changes are assumed to be unidirectional, and Dollo parsimony, where changes in one direction are assumed to occur only once in a tree. We refer to Felsenstein (Chapter 7) for a detailed description of these methods and others. In practice, these alternative parsimony approaches are rarely used.

1.4 Sankoff Optimization

All parsimony methods are special cases of a technique called *Sankoff optimization* (sometimes referred to as generalized parsimony or step matrix optimization). Sankoff optimization is based on a cost matrix $C = \{c_{ij}\}$, the elements of which define the cost c_{ij} of moving from a state i to a state j along any branch in the tree. The cost matrix is used to find the minimum cost of a tree and the set of optimal states at the interior nodes of the tree. Felsenstein gives an example of a cost matrix and the Sankoff downpass algorithm in his Figure 2.2 and the accompanying discussion (pp. 13-16). The uppass algorithm is presented in Chapter 6 (pp. 68-69 and Figs. 6.1 - 6.2).

A slightly different presentation of the Sankoff algorithm is given here. Assume that we are interested in a character with k states. For the downpass, assign to each node p a set $G_p = \{g_1^{(p)}, g_2^{(p)}, ..., g_k^{(p)}\}$ containing the (minimum) downpass cost $g_i^{(p)}$ of assigning state i to node p. We will use another similar set H_p, which will give the (minimum) cost of assigning each state i to the ancestral end of the branch having p as its descendant. The elements of H_p will be derived from G_p and C, as you might have guessed. Before we start the algorithm, we go through all terminal nodes and set $g_i^{(p)} = 0$ if the state i has been observed at tip p and $g_i^{(p)} = \infty$ otherwise. As usual, the downpass algorithm is formulated for a node p and its two descendant nodes q and r (Algorithm 5). At the root node ρ, we easily obtain the tree length l as $l = min_i(g_i^{(\rho)})$, the minimum cost for any state assignment. A worked example appears in Figure **??**.

The uppass is relatively straightforward. First, we find the state or states of the root node ρ associated with the minimum cost in the G_ρ set. Let F_ρ be the set of indices of these states. Now we can find the final state set of a node p from its downpass cost set G_p, the cost matrix C and the final state set F_a of its ancestor a easily (Algorithm 6).

In some cases, we may be interested not only in the optimal states at each interior node but also the cost of the suboptimal state assignments. To determine these, we need a slightly more complicated

Algorithm 5 Sankoff downpass algorithm

for all i **do**
  $h_i^{(q)} \leftarrow \min_j(c_{ij} + g_j^{(q)})$
  $h_i^{(r)} \leftarrow \min_j(c_{ij} + g_j^{(r)})$
end for
for all i **do**
  $g_i^{(p)} \leftarrow h_i^{(q)} + h_i^{(r)}$
end for

Algorithm 6 Algorithm for finding optimal state sets under Sankoff parsimony

$F_p \leftarrow \emptyset$
for all $i \in F_a$ **do**
  $m \leftarrow c_{i1} + g_1^{(p)}$
  for all $j \neq 1$ **do**
    $m \leftarrow \min(c_{ij} + g_j^{(p)}, m)$
  end for
  for all j **do**
    if $c_{ij} + g_j^{(p)} = m$ **then**
      $F_p \leftarrow F_p \cup j$
    end if
  end for
end for

algorithm. For this *Sankoff uppass algorithm*, let F_p be a set containing the final cost of assigning each state i to node p. We start the algorithm at the root ρ by setting $f_i^{(\rho)} = g_i^{(\rho)}$. The final cost set F_p of a node p can now be obtained from the cost matrix C, the final cost set F_a of its ancestor a and the downpass set H_p of the lower end of the branch connecting p with a (Algorithm 7). A worked example appears in Figure 6.

Algorithm 7 Sankoff uppass algorithm

for all j **do**
  $f_j^{(p)} \leftarrow min_i((f_i^{(a)} - h_i^{(p)}) + c_{ij} + g_j^{(p)})$
end for

1.5 Multiple Ancestral State Reconstructions

As we have already seen, it is common to have ambiguity concerning the optimal state at an interior node (that is, multiple states in the final state set). It does not take many ambiguous

internal nodes to result in a large number of equally parsimonious reconstructions of ancestral states for the entire tree. Felsenstein describes several methods for systematically resolving this ambiguity. Perhaps the most popular among these are the accelerated transformation (ACCTRAN) and delayed transformation (DELTRAN) methods. Both methods will select one state for each interior node among the most parsimonious assignments, resulting in a unique reconstruction of ancestral states. The selection is done using rules that differ between ACCTRAN and DELTRAN; in the former we are trying to press state changes towards the root of the tree, in the other we press the state changes towards the tips. In many cases, it is more appropriate to enumerate all possibilities or to randomly select among them than to systematically select a particular type of reconstruction. Felsenstein (Chapter 6) briefly describes how this can be accomplished and we refer to this chapter and the references cited there for more details.

1.6 Time Complexity

If we want to calculate the overall length (cost) of a tree with m taxa and n characters, each with k states, it is relatively easy to see that the Fitch and Wagner algorithms are of complexity $O(mnk)$ and the Sankoff algorithm is of complexity $O(mnk^2)$. If there is a small number of states, we can save a considerable amount of time in the Fitch and Wagner algorithms by packing several characters into the smallest unit handled by the processor, typically a 32-bit or 64-bit unit, and use binary operators to handle several characters simultaneously. There is also a number of shortcuts that can help us to quickly calculate the length of a tree if we know the length and, in particular, the final state sets for some other tree which is similar to the tree we are interested in. Felsenstein describes several of these techniques briefly but we find that the discussion is difficult to follow and it appears to be partly incorrect, so we recommend that you go to the original literature if you are interested in them. We will briefly touch on some of these computational shortcuts when we describe searches for most parsimonious trees later during the course.

1.7 Study Questions

1. What is the difference between Fitch and Wagner parsimony?
2. When Wagner parsimony is used for continuous characters on an interval scale (or raw measurements), what exactly is the amount of change that we are minimizing?
3. Find the Fitch uppass and downpass algorithms for an unrooted binary tree. Tip: formulate a variant of the algorithm for dealing with a basal trichotomy and use it when deriving the states for an interior node selected to be the 'root' of the unrooted tree.

4. Find the Fitch uppass and downpass algorithms for a polytomous tree. Tip: Use the same variants of the algorithms from the previous problem.

5. Express Fitch parsimony for a four-state character using a Sankoff cost matrix

6. Express Wagner parsimony for a four-state character using a Sankoff cost matrix

7. Is there a natural measure of branch length under Fitch parsimony? How would you calculate that branch length? Would the sum of all the branch lengths be equal to the tree length?

8. Can a Sankoff cost matrix have non-zero diagonal entries? Can it be non-metric (violate the triangle-inequality)?

Figure 5: A worked example of a Sankoff downpass algorithm. We start with a cost matrix (a) and then assign costs to the tips of the tree (b). For each node in the tree in a postorder traversal, we then calculate the cost of each state at the bottom end of each descendant branch (c). We then add these costs to get the costs at the top of the ancestral branch (d). Finally, at the root node, we find the minimum cost of the tree as the minimum cost of any state assignment (e).

Figure 6: A worked example of the Sankoff uppass algorithm. At the root node, the downpass has given us the final cost of all states. We can now work our way up the tree, one branch at a time, by subtracting the downpass costs at the bottom end of the branch from the final costs of the ancestor (a). This gives us the cost of each state at the bottom end of the branch given the subtree below the branch (b). The downpass costs of the descendant node similarly gives us the cost of each state in the tree above the branch. We then take each of the states of the descendant node and find the minimum combination of transformation cost plus downpass cost at descendant node plus remainder cost at the ancestor node (c). This results in the final costs for the descendant node (d).