Molecular Phylogenetics and Evolution
Vol. 22, No. 2, February, pp. 269–275, 2002
doi:10.1006/mpev.2001.1049, available online at http://www.idealibrary.com on IDEAL®

Optimization of Polytomies: State Set and Parallel Operations

Pablo A. Goloboff

Instituto Miguel Lillo, Miguel Lillo 205, 4000 San Miguel de Tucumán, Argentina

E-mail: instlillo@infovia.com.ar.

Received April 9, 2001; revised July 27, 2001

New algorithms for calculating the most parsimonious state sets for polytomies under Fitch parsimony are described. Because they are based on state set operations, these algorithms can be extended for optimization of several characters in parallel, thus increasing speed by a significant factor. This speed increase may facilitate analysis of molecular data sets, many of which contain hundreds of taxa, thousands of multistate nonadditive characters, and numerous polytomies.

INTRODUCTION

One of the schemes of substitution most commonly used in parsimony analysis of molecular sequences consists of considering all possible substitutions equally costly. This type of transformation cost is also used for many morphological characters. Under such a substitution scheme, the algorithms of Fitch (1971) and Hartigan (1973) allow calculating (for any given tree) both tree lengths and most parsimonious state sets (a process known as "optimization"). In computer programs for phylogenetic analysis the state sets are normally represented using individual bits of numbers, so that the state set operations (mostly unions and intersections) required to optimize a given character can be done more efficiently.

A now common technique to speed up optimization is to represent several characters in a single 32-bit word, long used in Hennig86 (Farris, 1988), and described by Ronquist (1998) and Moilanen (1999). This allows optimizing multiple characters together. The increase in speed can be very significant; for example, Rice *et al.* (1997) analyzed Chase *et al.*'s (1993) 500-taxon data set with PAUP for 3.5 months without ever finding minimum length, but by coupling efficient search strategies with parallel optimization, Goloboff (1999) was able to find minimum length trees (two steps shorter than the best trees Rice *et al.* found) in an average time of 10 min.

The algorithms of Fitch (1971), defined in terms of state set operations, can be easily adapted for parallel optimization. Fitch's algorithms, however, can be applied only to strictly dichotomous trees. The algorithms of Hartigan (1973) are more general, because they can be used for multifurcations as well. However, they require counting how many descendants have (or lack) each of the possible states for a given character, and thus they cannot be directly used in multicharacter optimization. The optimization of multifurcations is normally not required during tree-searches (the most time consuming part of parsimony analysis, which can use specific shortcuts; see Goloboff, 1994, 1996, 1999), but it is sometimes necessary to optimize an arbitrary set of trees. Molecular data sets often contain hundreds of taxa and thousands of characters. Therefore, even for tasks such as tree diagnosis or comparisons (relatively modest in comparison to a search for optimal trees), it may be necessary to optimize hundreds of thousands of trees with numerous polytomies. The computational demand for this task can be lessened by using better algorithms. The aim of this paper is thus to describe methods for optimizing polytomies, which are directly based on state set operations, and can thus be easily used in parallel optimization. The algorithms described are used in the program T.N.T. (Goloboff *et al.*, 1999). With some simplifications, the algorithms described would also be efficient for optimizing characters with very numerous states.

NOTATION

Throughout, the state sets of a node, of its left, medial, and right descendant, and of its ancestor are indicated as **n**, **l**, **m**, **r**, and **a**, respectively. A lowercase letter indicates a preliminary (locally optimal) state set, while an uppercase indicates a final (globally optimal) state set. The set of all possible states is defined as **P**. The complement of a set **S** is indicated as ~**S** (note that ~**S** must be a subset of **P**).

1055-7903/02 $35.00

ALGORITHMS FOR DICHOTOMOUS TREES

Multistate Characters

The algorithms of Fitch (1971) are defined almost exclusively in terms of state set operations. They determine preliminary states in a down-pass, and final states in an up-pass. In the down-pass, each node is assigned a state set **n** such that it minimizes the sum of transformation costs in the two branches subtended from the node:

$$\begin{array}{ll} \textbf{if } \mathbf{l} \cap \mathbf{r} \neq \varnothing & \mathbf{n} = \mathbf{l} \cap \mathbf{r} \\ \textbf{else} & \mathbf{n} = \mathbf{l} \cup \mathbf{r} \end{array}$$

In the up-pass, Fitch's algorithm determines **N** as follows:

if $\mathbf{A} \cap \sim\mathbf{n} = \varnothing$	$\mathbf{N} = \mathbf{A}$	**case 1**
if $\mathbf{A} \cap \sim\mathbf{n} \neq \varnothing$ **and** $\mathbf{n} = \mathbf{l} \cup \mathbf{r}$	$\mathbf{N} = \mathbf{n} \cup \mathbf{A}$	**case 2**
if $\mathbf{A} \cap \sim\mathbf{n} \neq \varnothing$ **and** $\mathbf{n} = \mathbf{l} \cap \mathbf{r}$	$\mathbf{N} = \mathbf{n} \cup (\mathbf{A} \cap (\mathbf{l} \cup \mathbf{r}))$	**case 3**

It is easy to see that, if no state absent from **n** is present in **A** (case 1), then for each of the states in **A** present in **n**, no extra cost will be incurred; if a state present in **n** but not in **A** is left in **N**, this would require one more step. For cases 2 and 3, the operations proposed by Fitch (1971) could be seen as simply adding to **n** those states shared in **A** and a state set which we could call **T** (for "temporary"). Thus:

$$\begin{array}{ll} \textbf{if } (\mathbf{l} \cap \mathbf{r} = \varnothing) & \mathbf{T} = \mathbf{P} \\ \textbf{else} & \mathbf{T} = \mathbf{l} \cup \mathbf{r} \end{array}$$

When so defined, **T** comprises all those states which, if assigned to the node instead of **n**, would require 0 or 1 steps beyond the minimum possible (see also Hartigan, 1973). When a state not present in **n** is found in **A**, this means that some reconstruction will have to postulate a transformation from one of the states in **A** to one of the states found in **n**, which is one step in the branch from ancestor to node, and no additional steps in the branch(es) from node to descendant(s). But, if some state(s) would require only one additional step in the branch(es) from node to descendant(s), and one less in the branch from ancestor to node, the total cost would be the same. This is why the state set **n** must often be enlarged to form **N** (Hartigan, 1973). If **T** is determined in this form during the down-pass itself (and stored), fewer operations are needed during the up-pass. The up-pass then simply becomes:

$$\begin{array}{ll} \textbf{if } \mathbf{A} \cap \sim\mathbf{n} = \varnothing & \mathbf{N} = \mathbf{A} \\ \textbf{else} & \mathbf{N} = \mathbf{n} \cup (\mathbf{A} \cap \mathbf{T}) \end{array}$$

Note that states can be eliminated from **n** to form **N** only when $\mathbf{A} \cap \sim\mathbf{n} = \varnothing$; otherwise, states can only be added to **n**, never eliminated from it, to form **N**.

Binary Characters

The algorithms shown above can be simplified when only two states are possible. In this case, when $\mathbf{l} \cap \mathbf{r} = \varnothing$, it follows that $\mathbf{n} = \mathbf{P}$ and (in the subsequent up-pass) $\mathbf{A} \cap \sim\mathbf{n}$ will always be $\varnothing$ (so that **T** is not actually required). The determination of **T** is therefore of interest only when $\mathbf{l} \cap \mathbf{r} \neq \varnothing$, and even then, only when $\mathbf{n} \neq \mathbf{P}$ (i.e., when **n** has a single state). In this case, it is easy to see that a state will require one additional step if present in one the descendant state sets and absent in the other (and the other state must be present in both **l** and **r**, because $\mathbf{l} \cap \mathbf{r} \neq \varnothing$). Then, in the case of binary characters, **T** can be determined during the down-pass simply as $\mathbf{T} = \mathbf{l} \cup \mathbf{r}$ (regardless of whether $\mathbf{l} \cap \mathbf{r} = \varnothing$).

ALGORITHMS FOR TRICHOTOMIES

Multistate Characters

When a node leads to three descendant branches, the assignments to **n** must be done considering the states in **l**, **m**, and **r**. The following is an exhaustive list of possibilities during the down-pass:

Case (1): If some states are shared among the three descendants of the node then those states must be assigned to **n** and no steps are required.

Case (2): Otherwise, if any two out of **l**, **m**, and **r** share states, then assigning to **n** any of the states present in two out of the three descendant branches will require one step, the minimum possible.

Case (3): Otherwise, any of the states present in **l**, **m**, or **r** will require two steps, the minimum possible.

T can be determined as follows. For case (1), any state present in two out of the three descendants, but not in the three, will require (when assigned to the node) one more step than any state shared by the three descendants. For case (2), any state present in only one of **l**, **m**, and **r** will require two steps, instead of the single step required by the states shared by two out of the three. Last, for case (3), any of the states in **P** not present in **l**, **m**, or **r** will require three steps, instead of the two required by any of the states present in **l**, **m**, or **r**. Thus,

$$\textbf{if } \mathbf{l} \cap \mathbf{m} \cap \mathbf{r} \neq \varnothing \begin{cases} \mathbf{n} = \mathbf{l} \cap \mathbf{m} \cap \mathbf{r} \\ \textbf{no length increase} \\ \mathbf{T} = (\mathbf{l} \cap \mathbf{m}) \cup (\mathbf{l} \cap \mathbf{r}) \cup (\mathbf{r} \cap \mathbf{m}) \end{cases}$$

$$\textbf{else} \begin{cases} \textbf{if } (\mathbf{l} \cap \mathbf{m}) \cup (\mathbf{l} \cap \mathbf{r}) \cup (\mathbf{r} \cap \mathbf{m}) \neq \varnothing \begin{cases} \mathbf{n} = (\mathbf{l} \cap \mathbf{m}) \cup (\mathbf{l} \cap \mathbf{r}) \cup (\mathbf{r} \cap \mathbf{m}) \\ \textbf{increase length in 1} \\ \mathbf{T} = \mathbf{l} \cup \mathbf{m} \cup \mathbf{r} \end{cases} \\ \textbf{else} \quad \begin{cases} \mathbf{n} = \mathbf{l} \cup \mathbf{m} \cup \mathbf{r} \\ \textbf{increase length in 2} \\ \mathbf{T} = \mathbf{P} \end{cases} \end{cases}$$

When **T** is determined in this way, **N** can be calculated during the up-pass just as before, without regard for whether the node is a bifurcation or a trifurcation:

$$\begin{array}{ll} \textbf{if } \mathbf{A} \cap \sim\mathbf{n} = \varnothing & \mathbf{N} = \mathbf{A} \\ \textbf{else} & \mathbf{N} = \mathbf{n} \cup (\mathbf{A} \cap \mathbf{T}) \end{array}$$

Binary Characters

The determination of **n** and **T** can be simplified in the case of binary characters. Case (1) remains the same, but as soon as the conditions for case (1) do not hold, it follows that the conditions for case (2) must hold (i.e., case (3) is impossible for binary characters). Thus, for binary characters,

$$\textbf{if } \mathbf{l} \cap \mathbf{m} \cap \mathbf{r} \neq \varnothing \begin{cases} \mathbf{n} = \mathbf{l} \cap \mathbf{m} \cap \mathbf{r} \\ \textbf{no length increase} \\ \mathbf{T} = (\mathbf{l} \cap \mathbf{m}) \cup (\mathbf{l} \cap \mathbf{r}) \cup (\mathbf{r} \cap \mathbf{m}) \end{cases}$$

$$\textbf{else} \begin{cases} \mathbf{n} = (\mathbf{l} \cap \mathbf{m}) \cup (\mathbf{l} \cap \mathbf{r}) \cup (\mathbf{r} \cap \mathbf{m}) \\ \textbf{increase length in 1} \\ \mathbf{T} = \mathbf{l} \cup \mathbf{m} \cup \mathbf{r} \end{cases}$$

MULTICHARACTER ALGORITHMS

Representation

The state sets are normally represented in computer programs as sets of ON/OFF bits. Thus, the state set {0, 3, 4} will be the number $2^0 + 2^3 + 2^4 = 25$. Using, for example, the C programming language, it then becomes possible to make the state set operations by using the bitwise operations "and" (&, intersection), "inclusive or" (|, union), "exclusive or" (^, a bit becomes OFF if either ON in both or OFF in both, ON otherwise), and "complement" (~, a bit becomes ON if OFF, and vice versa). If the state sets for several characters are represented in a single number, it then becomes possible to optimize several characters at a time (Farris, 1988; Ronquist, 1998; Moilanen, 1999). Two possibilities arise here: all the states for several characters can be included in bit fields of a single number ("horizontal packing" of Ronquist, 1999), or different states may be represented by different numbers and every bit of a number represents a given character ("vertical packing"). The examples below discuss only horizontal packing, but the method could be adapted for vertical packing as well. For parallel optimization, some of the operations require right (≫) or left (≪) shifting of the bits. In this section, $\mathbf{S_i}$ indicates the number corresponding to several state sets **i** combined ("horizontally") in a single 32-bit word (obviously, real programs will use pointers for this).

Multistate Characters

Ronquist (1998) proposed down- and up-pass algorithms for both vertical and horizontal packing, and for four-state characters, for bifurcations only. For the sake of comparison, the following pseudocode illustrates a down-pass for horizontally packed four-state characters, in the case of a bifurcation:

```
x = Sl & Sr;
y = Sl|Sr;
c = F0 ^ ((F0 & x)|((F1 & x) ≫ 1)
      |((F2 & x) ≫ 2)|(F3 & x) ≫ 3));
length +=
      onbits [(c|(c ≫ 15)) & 65535];
c | = (c ≪ 1)|(c ≪ 2)|(c ≪ 3);
ST = c|y;
Sn = x|(y & c);
```

The constants **F0–F3** are defined as masks with (respectively) every first, second, third, and fourth bit of

each field as ON (e.g., **F0** = 286331153). The code for bifurcations in the example above differs from Ronquist's (1998) algorithms in three respects: the determination of **c**, the calculation of the length increase, and the calculation of a temporary value $\mathbf{S_T}$ to be later used in the up-pass. The variable **c** is first assigned a value such that the first bit of a field is ON if the bit field is empty (OFF otherwise), and then a value such that a bit field is full (i.e., equals **P**) if the first bit of the field was ON (empty otherwise). Ronquist (1999) proposed using a loop through the bit fields, shifting a mask; using constants improves speed. (But it is possible to make further improvements so that similar results are achieved with about half the operations (Farris, pers. comm.); the example above, however, suffices to illustrate the parallelization.) The length increase in Ronquist's (1998) algorithms was also made using a loop through the bit fields, while the above example uses Moilanen's (1999) approach, precalculating the numbers of ON bits in a lookup table (stored in the array onbits). The down-pass for sets of four-state characters in the case of trifurcations would be

```
x = Sl & Sm & Sr;
y = (Sl & Sm)|(Sl & Sr)|(Sm & Sr);
z = Sl|Sm|Sr;
c = F0 ^ ((F0 & x)|((F1 & x) >> 1)
    |((F2 & x) >> 2)|(F3 & x) >> 3));
d = F0 ^ ((F0 & y)|((F1 & y) >> 1)
    |((F2 & y) >> 2)|(F3 & y) >> 3));
length +=
    onbits [(c|(c >> 15)) & 65535]+
    onbits [(d|(d >> 15)) & 65535];
c |= (c << 1)|(c << 2)|(c << 3);
d |= (d << 1)|(d << 2)|(d << 3);
ST = y|(c & z)|d;
Sn = x|(y & c)|(z & d);
```

Having determined $\mathbf{S_T}$ as shown, the up-pass can proceed independent of whether the node is bi- or trifurcated:

```
x = SA & ~Sn;
c = (F0 & x)|((F1 & x) >> 1)
    |((F2 & x) >> 2)|(F3 & x) >> 3);
c |= (c << 1)|(c << 2)|(c << 3);
SN = (SA & ~c)|(c & (Sn|SA & ST));
```

Binary Characters

The algorithms for both bi- and trifurcations can be simplified in the case of binary characters. In this case, **B0** and **B1** are defined as masks with every first and second bit ON, respectively. The following pseudocode will optimize 16 characters in tandem:

```
if (bifurcation){
    x = Sl & Sr;
    c = B0 ^ ((x & B0)|((x & B1) >> 1));
    length +=
        onbits [(c|(c >> 15)) & 65535];
    c |= (c << 1);
    ST = Sl|Sr;
    Sn = x|c;}
else if (trifurcation){
    x = Sl & Sm & Sr;
    y = (Sl & Sm)|(Sl & Sr)|(Sm & Sr);
    c = B0 ^ ((x & B0)|((x & B1) >> 1));
    length +=
        onbits [(c|(c >> 15)) & 65535];
    c |= (c << 1);
    ST = y|(c & (Sl|Sm|Sr));
    Sn = x|(y & c);}
```

The up-pass is similar to the four-state case:

```
x = SA & ~Sn;
c = (B0 & x)|((B1 & x) >> 1);
c |= (c << 1);
SN = (SA & ~c)|(c & (Sn|SA & ST));
```

Multifurcations

When the node leads to more than three descendant nodes, the operations to determine $\mathbf{S_n}$ and $\mathbf{S_T}$ are more involved. For each field (in the horizontal packing) it becomes necessary to count how many descendants have a given state as present and how many as absent. It is possible to do this without explicitly counting the bits in each of the descendants. Table 1 shows an example of code, for four-state characters (binary characters would require using masks **B0** and **B1** instead of **F0–F3**). To count in parallel the number of descendants with each state (of each character) as absent, an

TABLE 1

Code for Assigning Preliminary Sets of States in Parallel, for Multifurcations, When Node *node* (Leading to *numdes* Descendants, the Leftmost of Which Is *first_desc [node]*) Is to Be Optimized

```
 1  i = first_desc [node];
 2  for (c = 0; c ++ <numdes;) absin [c] = 0;
 3  absin [0] = ~0;
 4  k = 2;
 5  while (i >= 0){
 6    x = ~pckmat [i];
 7    i = sister [i];
 8    for (j = k ++; j --> 1;)
 9      absin [j] |= absin [j - 1] & x;}
10  msk = absin [numdes];
11  for (c = numdes; c --;){
12    absin [c] = (absin [c]|msk) ^ msk;
13    msk |= absin [c];}
14  bak = 0;
15  this = x = absin [0];
16  msk = F0^((F0 & x)|((F1 & x) >> 1)
        |((F2 & x)>> 2)|(F3 & x) >> 3));
17  length += onbits [(msk|(msk >> 15)) & 65535];
18  msk |= (msk << 1)|(msk << 2)|(msk << 3);
19  prevmsk = ~0;
20  for (c = 0; c ++< numdes;){
21    x = absin [c];
22    this |= msk & x;
23    bak |= prevmsk & x;
24    prevmsk &= msk;
25    if (!prevmsk) break;
26    curmsk = F0^((F0 & x)|((F1 & x) >> 1)
        |((F2 & x)>> 2)|(F3 & x) >> 3));
27    curmsk |= (curmsk << 1)|(curmsk << 2)|(curmsk << 3);
28    msk &= curmsk;
29    x = msk & F0;
30    length += onbits [(x|(x >> 15)) & 65535];}
31  pckmat [node] = this;
32  pckbak [node] = bak;
```

Note. The count of descendants not having a given state is stored in **absin** (if a bit of **absin[i]** is ON; this means that **i** descendants lack the corresponding state). The sister of node **i** is **sister[i]** (**sister[i]** equal to −1 indicates that **i** is the last descendant of **node**). The variable **pckmat[i]** represents the (super)sets of characters for node **i**, the variable **pckbak[i]** represents the (super)sets of states suboptimal by one step for node **i** (to be used in the up pass).

array of integers (**absin**) is used. This array of integers is first set to a preliminary value such that, if **j** descendants have the corresponding bit as OFF, the values for any $\mathbf{i} \leq \mathbf{j}$ will have the bit as ON (this is done in the loop of lines 5–9). Then, the final values are calculated; for the **ith** value, an ON bit is changed to OFF if the same bit is ON in the value for $\mathbf{i + 1}$ (this is done in the loop of lines 11–13). Figure 1 shows an example of the values of the preliminary and final values of the bit counters, with five descendants. Once the bit counters have been set, two masks must be initialized (Fig. 2). The first, **msk**, is set as full (lines 15–18) for all those fields for which no bit (=state) is shared by all descendants (in the example, this is all fields, except the last one; see Fig. 2). The second, **prevmsk**, is always initialized as full for every field (line 19). The bit counters for 0, 1, 2, $\cdots$ n (where n = number of descendants) are then visited in order. Every time a field has nonzero value, the corresponding states are added to the downpass states for the node (line 22; this will only add states if no states had been placed in the field before, because in that case the mask will be empty for the

```
Descendants


   0100 1000 0011 0101 ( first descendant )
   0010 0101 0010 1111 ( second      "     )
   0100 1000 0100 0101 ( third       "     )
   1000 1000 0001 1100 ( fourth      "     )
   0100 0001 0100 1100 ( fifth       "     )


Preliminary values of count:


   1111 1111 1111 1111 ( absin [ 0 ] )
   1111 1111 1111 1011 ( absin [ 1 ] )
   1111 1111 1111 0010 ( absin [ 2 ] )
   1011 0111 1111 0010 ( absin [ 3 ] )
   1011 0110 1000 0000 ( absin [ 4 ] )
   0001 0010 1000 0000 ( absin [ 5 ] )


Final values of count:


   0000 0000 0000 0100 ( absin [ 0 ] )
   0000 0000 0000 1001 ( absin [ 1 ] )
   0100 1000 0000 0000 ( absin [ 2 ] )
   0000 0001 0111 0010 ( absin [ 3 ] )
   1010 0100 0000 0000 ( absin [ 4 ] )
   0001 0010 1000 0000 ( absin [ 5 ] )
```

FIG. 1. Example of determination of preliminary and final values for the array of bit counters (see text for explanation), for a node leading to five descendants, and for four-state characters (only four bit fields are shown).

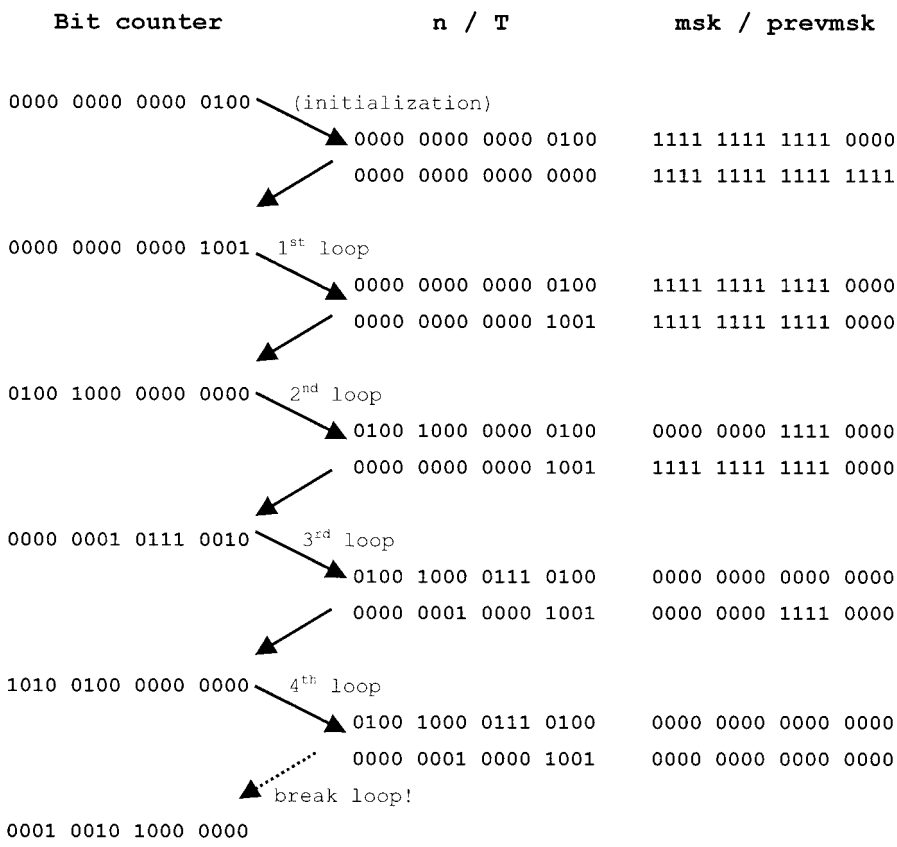

FIG. 2. Determination of preliminary and back-up states, based on the final values of the bit counters calculated in Fig. 1.

field), and the mask for the field is set as empty (lines 26–28). When a field had a nonzero value in the previous bit counter, the states for the current bit counter (if any) are added to the backup (temporary) states (line 23), and **prevmsk** is set as empty for the field (line 24). Note that for each bit field that becomes nonempty, **msk** is set as empty, and **prevmsk** is set as nonempty in the next execution of the loop. Once all the fields have been set for both the locally optimal and backup states, **prevmsk** will be 0, and the loop can be interrupted (line 25; see Fig. 2).

Given that (for a character with **s** states, and a node leading to **d** descendants) a state occurring in less than **d/s** descendants can never be in the most parsimonious preliminary state set of the node, and a state occurring in less than (**d/s**) − **1** descendants can never be in the set of backup states, it would appear that the code shown could be improved by taking this into account, changing the limits for the loop of lines 8–9, so that **k** never increases beyond (**d/s**) + **1**. The amount of time saved, however, would not be very significant, because **k** already remains below (**d/s**) + **1** for a fraction (**s** − **1**)/**s** of the time. Thus, for four-state characters, this would save at most 25% of the execution time of the loop of lines 8–9 for about 25% of the loop iterations; the entire loop, in turn, uses only about 25% of the total execution time during optimization. Thus, for four-state characters, execution would be speeded up by less than 2%. For two-state characters, the increase would be somewhat greater, but still well below 5%.

TABLE 2

Times (in s) to Optimize 300 Random Trees for Chase *et al.*'s (1993) 500-Taxon Data Set

	Trichotomies		Polytomies	
Single character algs.	34.4	60.6	40.3	51.0
PAUP*	14.0	27.0	16.5	43.0
T.N.T.	3.2	4.1	5.8	6.3

Note. The polytomous trees had up to 12-tomies.

Speedups

The gains in speed by applying the algorithms described can be very significant. As an example, Table 2 shows the times to do a down-pass (e.g., to calculate length) and to do a two-pass optimization (e.g., to collapse zero-length branches) on 300 randomly generated trees (on a 266-MHz Pentium II machine, running under Windows NT), for Chase *et al.*'s (1993) 500-taxon

data set. It can be seen in Table 2 that applying the algorithms for trifurcations described here makes the down-pass 10.7 times faster, and the double-pass optimization about 14.7 times (this data set has over 75% of the characters with three or four states; the parallelization algorithms for trifurcations save even more time for binary characters, because instead of optimizing eight characters in parallel, they optimize 16). For higher furcations, the increase in efficiency by using the parallelization described here is, if not so large, also very significant, speeding up the down-pass by a factor of 6.9 and the double-pass by a factor of 8.1. Note that the trees used to evaluate the method had no node leading to more than 12 descendants; the amount of time saved by the algorithms (at a given node) depends on the number of descendants of the node; for many more descendants, less time could be saved (the loop of lines 5–9 would use more time). The widely used PAUP* (Swofford, 1998) may implement some kind of (undescribed) parallelization for multifurcations, because it is over two times faster than the single character algorithms; the algorithms, however, are significantly slower than the ones used by TNT (4.3 to 6.3 times slower, in the case of trifurcations, and 2.8 to 6.8 times slower, in the case of higher furcations).

ACKNOWLEDGMENTS

I thank Rob DeSalle, Steve Farris, Julián Faivovich, Gonzalo Giribet, and Mike Steel for discussion, comments, and/or facilitating literature. The research was carried out with deeply appreciated support from CONICET (PEI 0324/97) and Agencia Nacional de Promoción Científica y Tecnológica (PICT 98 01-04347), in the facilities provided by the Instituto Superior de Entomología "Dr. Abraham Willink."

REFERENCES

Chase, M. W., Soltis, D. E., Olmstead, R. G., Morgan, D., Les, D. H., Mishler, B. D., Duvall, M. R., Price, R. A., Hills, H. G., Qiu, Y.-L., Kron, K. A., Rettig, J. H., Conti, E., Palmer, J. D., Manhart, J. R., Sytsma, K. J., Michaels, H. J., Kress, W. J., Karol, K. G., Clark, W. D., Hedren, M., Gaut, B. S., Jansen, R. K., Kim, K.-J., Wimpee, C. F., Smith, J. F., Furnier, G. R., Strauss, S. H., Xiang, Q.-Y., Plunkett, G. M., Soltis, P. S., Swensen, S. M., Willimas, S. E., Gadek, P. A., Quinn, C. J., Eguiarte, L. E., Golenberg, E., Learn, Jr., G. H., Graham, S. W., Barret, S. C. H., Dayanandan, S., and Albert, V. A. (1993). Phylogenetics of seed plants: An analysis of nucleotide sequences from the plastid gene rbcL. *Ann. Mo. Bot. Gard.* **80:** 528–580.

Farris, J. (1988). Hennig86. Program and documentation, distributed by the author.

Fitch, W. (1971). Toward defining the course of evolution: Minimal change for a specific tree topology. *Syst. Zool.* **20:** 406–416.

Goloboff, P. (1994). Character optimization and calculation of tree lengths. *Cladistics* **9:** 433–436.

Goloboff, P. (1996). Methods for faster parsimony analysis. *Cladistics* **12:** 199–220.

Goloboff, P. (1999). Analyzing large data sets in reasonable times: Solutions for composite optima. *Cladistics* **15:** 415–428.

Goloboff, P., Farris, J., and Nixon, K. (1999). T.N.T.: Tree Analysis Using New Technology. Program available at www.cladistics.com.

Hartigan, J. (1973). Minimum mutations fit to a given tree. *Biometrics* **29:** 53–65.

Moilanen, A. (1999). Searching for most parsimonious trees with simulated evolutionary optimization. *Cladistics* **15:** 39–50.

Rice, K., Donoghue, M., and Olmstead, R. (1997). Analyzing large data sets: rbcL 500 revisited. *Syst. Biol.* **46:** 554–563.

Ronquist, F. (1998). Fast Fitch-parsimony algorithms for large data sets. *Cladistics* **14:** 387–400.

Swofford, D. (1998). PAUP*: Phylogenetic Analysis Using Parsimony (and other methods), version 4.0b4a. Sinauer Associates, Sunderland, MA.