Column Sorting: Rapid Calculation of the Phylogenetic Likelihood Function

SERGEI L. KOSAKOVSKY POND1,3 AND SPENCER V. MUSE2

1Department of Mathematics, University of Arizona, Tucson, Arizona 85721, USA
2Bioinformatics Research Center and Department of Statistics, North Carolina State University, Raleigh, North Carolina 27695-7566, USA;
E-mail: muse@stat.ncsu.edu
3Current Address: Antiviral Research Center, University of California, San Diego, California, USA

Abstract.—Likelihood applications have become a central approach for molecular evolutionary analyses since the first computationally tractable treatment two decades ago. Although Felsenstein’s original pruning algorithm makes likelihood calculations feasible, it is usually possible to take advantage of repetitive structure present in the data to arrive at even greater computational reductions. In particular, alignment columns with certain similarities have components of the likelihood calculation that are identical and need not be recomputed if columns are evaluated in an optimal order. We develop an algorithm for exploiting this speed improvement via an application of graph theory. The reductions provided by the method depend on both the tree and the data, but typical savings range between 15% and 50%. Real-data examples with time reductions of 80% have been identified. The overhead costs associated with implementing the algorithm are minimal, and they are recovered in all but the smallest data sets. The modifications will provide faster likelihood algorithms, which will allow likelihood methods to be applied to larger sets of taxa and to include more thorough searches of the tree topology space. [DNA sequence; graph theory; likelihood; phylogeny; traveling salesman problem.]

The use and influence of likelihood methods in molecular evolution has increased dramatically over the past decade, largely because of improvements in affordable desktop computing speeds. Nonetheless, likelihood methods are still slow in comparison to other methods, notably parsimony and distance methods for phylogenetic estimation. In this paper we provide a simple method for markedly reducing the cost of likelihood calculations. The approach has low memory and implementation requirements, and almost universally recovers any preprocessing overhead costs associated with using the method. Improvements such as this one are important to the end user because they allow likelihood methods to be applied to larger data sets. In addition, in cases where searches through tree topologies are required, these types of improvements allow a larger number of trees to be searched in a given amount of time. Because likelihood methods typically have good statistical properties, any algorithmic advances that lead to more widespread use of likelihood are important.

THE PRUNING ALGORITHM AND MOTIVATION FOR COLUMN SORTING

To motivate the method presented here, it is necessary to describe the pruning algorithm for computing likelihood functions for homologous molecular sequences related by a phylogenetic tree. For a complete discussion refer to Felsenstein (1981). For the moment, consider only the likelihood calculation for position c in a multiple sequence alignment (i.e., a single column), and let the likelihood for that position be denoted $L_c$. We assume the typical case where the characters (e.g., nucleotides or amino acids) at the leaves of a tree $r$ are observed in the data, whereas the character states at the internal nodes are unknown. Let there be $N$ possible states ($N = 4$ for nucleotide sequences, $N = 20$ for amino acid sequences) with equilibrium frequencies $\pi_i, i = 1, 2, \ldots, N$, and let $p_{ij}(n)$ denote the transition probability for changing from state $i$ to state $j$ along the branch terminating at node $n$. The transition probabilities and equilibrium frequencies are properties of the model of sequence change selected for the data analysis, (e.g., Kimura, 1980; Hasegawa et al., 1985).

As shown in Felsenstein (1981), the likelihood is a function of the observed states, the transition probabilities for all branches, and the equilibrium frequencies of the states. To reveal the recursive nature of computing the likelihood function, introduce the partial likelihood $L_c(n)$, which is the likelihood for the subtree rooted at node $n$ given that the state at node $n$ is $i$. Felsenstein’s pruning algorithm is based on exploiting properties of the partial likelihoods to obtain a fast recursion, and the following facts can be found in Felsenstein (1981):

- If $n$ is a leaf then $L_c(n) = 0$ if $i$ is not the observed state, and $L_c(n) = 1$ otherwise. (Sequence data often contain ambiguities or gaps. The definition of the likelihood at a leaf can be extended to accommodate these as well, but we will ignore that complication for sake of brevity).
- If nodes $\lambda_d, d = 1 \ldots D$ are the immediate descendants of an internal node $n$ then $L_c(n) = \prod_{d=1}^{D} \sum_{j=1}^{N} p_{ij}(\lambda_d) L_c(\lambda_d)$.
- If $r$ is the root node of the tree, then the likelihood for the entire tree is $L(r) = \sum_{i=1}^{N} \pi_i L_c(r)$

The pruning algorithm offered the first substantial computational improvement for evaluating the likelihood function, taking advantage of the recursive nature of the function. The likelihood for the entire tree is computed by calculating the partial likelihoods starting at the leaves and working towards the root. For the tree in Figure 1,
the partial likelihoods are computed in the following order: 1, 2, 8, 3, 10, 4, 5, 6, 9, 7, 11, 12, consistent with the postorder tree traversal order. The internal nodes are emphasized in bold, and partial likelihood computations at these nodes involve summations.

When likelihood functions for data sets are computed, the same tree and transition probabilities $p_{ij}(n)$ are typically used at each column. The same series of partial likelihood calculations are performed in the same order, differing only in the values at the leaves (i.e., different columns in the multiple sequence alignment). The total likelihood is then found by taking the product of all the individual site likelihoods, $L(\tau) = \prod_{c=1}^{C} L_c$, where $C$ is the number of columns in the alignment.

Consider computing the likelihood function for the tree and data found in Figure 1. In a naive implementation, 6 pruning algorithm passes are needed to evaluate the likelihood function, 1 for each alignment column, resulting in a total of 30 partial likelihood calculations at the internal nodes. Notice that the total likelihood does not depend on the order in which the likelihoods of sites are computed. However, site ordering does affect the number of partial likelihood evaluations. When the likelihood for the first site is found, all of the partial likelihoods need to be recomputed; when we move to the next site, it would be desirable to reuse some of the partial likelihoods from the previous step if possible. The partial likelihood for node $n$ can be reused (i.e., its value is unchanged from that of the previously computed column) whenever all the leaves that are descendants of node $n$ are identical to the corresponding leaves at the prior site. For example, in moving from site 1 to site 2 in the sample data, all the leaves are changed except for leaf 7. Thus, partial likelihoods for all internal nodes must be recomputed. However, site ordering does affect the number of partial likelihood evaluations for internal nodes, a savings of 2 partial likelihood evaluations.

Programmers have long recognized that identical columns have the same likelihood and only need to be computed once. Felsenstein calls this process aliasing in his program documentation (Felsenstein, 1993). However, it is clear from the data that sites 1 and 5 are quite similar, and the number of partial likelihood evaluations could be decreased by rearranging the sites as 1, 5, 2, 3, 6, 4. Following this reordering, in going from column 1 to column 5 the partial likelihoods at only two internal nodes (11 and 12) must be recomputed, and only three (9, 11, and 12) are updated upon going from column 3 to column 6. The total number of internal node computations for the entire data set is now 25, a reduction of 17% from the original 30 computations. Real sequence alignments usually consist of hundreds or thousands of columns, and the trees can grow to have hundreds of leaves. For a large data set it would clearly be quite beneficial to arrange the sites in such a way as to maximize the number of reusable partial likelihoods, if such an arrangement can be found without excess computation.

**COLUMN SORTING**

Consider a fixed tree and a matrix of transition probabilities for each branch, with the transition probabilities assumed equal across all $C$ alignment columns (i.e., no site-to-site rate heterogeneity.) Denote the set of characters at alignment position $c$ as $s_c$, for $1 \leq c \leq C$. The question of optimal ordering of the columns can now be rephrased as “find the permutation of indices $c$, so that the number of partial likelihood evaluations for the internal nodes is minimized.” As we will show, this question is reducible to finding the shortest Hamiltonian path in the complete Euclidean graph (i.e., the traveling salesman problem).

**A Metric for State Vectors**

Define $S^N_M$ to be the set of vectors of length $M$ with integer entries taking values between 1 and $N$. This set can also be thought of as the set of strings of length $M$ over an alphabet with $N$ characters. In the current context, $M$ is the number of sequences in the alignment, whereas $N$ is the number of possible characters ($N = 4$ for nucleotide sequences). For a given tree $\tau$ with $M$ leaves and two observed state vectors $s_1$ and $s_2$ from $S^N_M$, we agree to call an internal node $n$ of the tree $\tau$ tainted with respect to $(s_1, s_2)$ if when we consider the subtree defined by internal node $n$, the leaves in columns $s_1$ and $s_2$ are not identical in the subtree.

The distance function $d_\tau(s_1, s_2)$ is defined for any two state vectors of length $M$ as

$$d_\tau(s_1, s_2) = \sum_{\text{tainted nodes } n} \text{number of children of } n$$

Note that for strictly bifurcating trees

$$d_\tau(s_1, s_2) = 2 \times (\text{number of tainted nodes})$$
and for unrooted bifurcating trees, which are typical in phylogenetic analyses,
\[
d_{r}(s_1, s_2) = 2 \times (\text{number of tainted nodes}) + 1 \quad (2)
\]
where the additional term arises from the fact that the root node in an unrooted tree has three children. Intuitively, the metric is simply the number of branches in the tree for which partial likelihoods from \(s_1\) cannot be reused when evaluating the likelihood of \(s_2\). The calculation of \(d_{r}(s_1, s_2)\) can be accomplished by one postorder traversal of the tree and thus in time \(O(M)\), recalling that \(M\) is the number of aligned sequences. Refer to Algorithm 1 in the Appendix for one possible algorithm.

Reduction to a Graph Traversal Problem

Let us return to the example of the opening section and rephrase the problem of optimal column ordering in graph theoretical terms. Construct the complete graph \(G\) (which has a graph-theoretical name, \(K_6\), with vertices corresponding to the columns (state vectors) in the sequence alignment and the length of the edge between two vertices \(s_i\) and \(s_j\) given by \(d_r(s_i, s_j)\). The distances between vertices (columns of data) are collected in the following triangular matrix:

\[
\begin{array}{cccccc}
 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 \\
n_1 & 11 & 11 & 9 & 5 & 11 \\
n_2 & 9 & 11 & 11 & 9 \\
n_3 & 11 & 11 & 7 \\
n_4 & 9 & 11 \\
n_5 & 11 \\
n_6 & \\
\end{array}
\]

The distances from this table can be easily related to the “costs” used in the opening example by means of Equation 2.

The task of computing the likelihood of all columns can be thought of as the task of traversing the graph \(G\) in Figure 2, visiting each vertex exactly once (i.e., traversing a Hamiltonian path). The total length of the path indicates the total number of partial likelihood calculations, and we seek to minimize it. Any permutation of column indices defines a new path in the graph in an obvious way. For example, the length of the “natural” path
\[
s_1 \mapsto s_2 \mapsto s_3 \mapsto s_4 \mapsto s_5 \mapsto s_6
\]
is
\[
d_r(s_1, s_2) + d_r(s_2, s_3) + d_r(s_3, s_4) + d_r(s_4, s_5) + d_r(s_5, s_6) = 11 + 9 + 11 + 9 + 11 = 51.
\]
On the other hand, the path
\[
s_1 \mapsto s_5 \mapsto s_4 \mapsto s_3 \mapsto s_6 \mapsto s_2
\]
has length 5 + 9 + 11 + 7 + 9 = 41.

An Algorithm for Finding a Suboptimal Hamiltonian Path

The traveling salesman problem (TSP) is solved by finding the minimal Hamiltonian cycle in the graph. TSP has a history of interest to researchers working in combinatorial optimization. Unlike other problems whose general solutions were developed in the theory of linear programming during the early 1950s, the TSP was stubborn in its worst-case difficulty and was thought to be fundamentally hard. This suspicion was confirmed shortly after the definition of an equivalence class of NP-hard problems known as NP-complete in the early 1970s and the subsequent identification of the TSP as an NP-complete problem (Karp, 1972). Fortunately, many approximating methods and techniques for “good” suboptimal solutions to the TSP have been identified. A well known approximation technique uses the minimal spanning tree (MST) and the triangle inequality to construct a cycle that is within a factor of two of the minimal Hamiltonian cycle. The proof of this assertion can be found in many graph theory books. An accessible presentation may be found, for instance, in Gibbons (1985). Refer to Algorithm 2 in the Appendix for details. Prim’s algorithm for finding a MST has running time \(O(S^2)\), where \(S = |G|\) is number of columns in the data. There are several alternative algorithms for constructing a MST; however, none of them offer a speed advantage over Prim’s algorithm when applied to a complete graph (Cheriton and Tarjan, 1972). Note that there is a well known approximate algorithm for solving TSP on graphs whose edge lengths satisfy the triangle inequality (Christofides, 1976), which gets within a factor of 3/2 of the optimal solution, and it can be used instead of the above algorithm. However, the overhead required of that method is \(O(S^3)\) and is prohibitive for data sets with many sequences.

An application of Prim’s algorithm to the graph \(G\) in Figure 2 starting with vertex 1 yields the MST pictured in Figure 3. The suboptimal path constructed by using this tree is
\[
1 \mapsto 5 \mapsto 4 \mapsto 3 \mapsto 6 \mapsto 2
\]
of total length 41, a computational reduction of approximately 20% from the unsorted data value of 51. For this simple example, it can be shown by exhaustive search that this path is actually an optimal one.

**The Modified Pruning Algorithm**

In order to make use of the column sorting process, a few slight modifications must be made to the pruning algorithm. First of all, all partial likelihoods of internal nodes must be stored between the evaluations of two successive columns. This requirement isn’t too stringent, because the pruning algorithm itself almost requires it. Although it is possible to store only a fixed number of partial likelihoods for an evaluation at a given column (that number will depend on the maximal number of children at any given node, but not on the size of the tree), the memory requirement for storing conditional likelihoods is dwarfed by the amount of space needed to store transition matrices at each node. Additionally, each internal node must be equipped with a boolean flag, indicating whether the transition from one site to the next in the data has tainted the node. There is a minimal amount of bookkeeping involved, but its cost is negligible relative to the cost of likelihood evaluation. The pseudocode fragment in Algorithm 3 of the Appendix illustrates the logic of the modifications that must be made to accommodate column sorting. The use of column aliasing is assumed, so the set of columns to be sorted consists only of one copy of each column type found in the data.

The algorithm suggested above has computational complexity of \( O(M|G|^2) \), where \(|G|\) is the number of vertices in the graph \( G \) (i.e., the number of distinct column types in the data) and \( M \) is the number of leaves in the tree. The factor \( M \) appears because each distance computation entails comparing values at each leaf of the tree, and if necessary, traversing the tree upwards from a leaf to the root. Storage requirements for the column sorting are of order \( O(|G|^2) \), and the modified pruning algorithm only requires an extra boolean flag per in each internal node, and thus is \( O(M) \).

**Further Time Saving Heuristics**

A substantial portion of time in execution of the pruning algorithm is spent traversing the tree, and further time savings can be realized by trimming the number of nodes that must be traversed for each site. This is especially noticeable for bifurcating trees on nucleotide data, where each partial likelihood evaluation requires at most 15 floating point operations (23 at the root, if the tree was unrooted), but because the reduction in traversal time offered below takes very little effort and overhead, it is worthwhile to implement for all molecular data types.

Consider our example data set with the columns ordered as

\[
\begin{align*}
s_1 & \mapsto s_5 \mapsto s_4 \mapsto s_3 \mapsto s_6 \mapsto s_2.
\end{align*}
\]

Moving from site 1 to site 5, only the last leaf changes, so the traversal of the tree in the pruning algorithm can be started at that last leaf (because everything else in the tree is unchanged). Thus instead of spending time traversing all seven leaves and five internal nodes, we only need traverse leaf 7 and internal nodes 11 and 12, i.e., only 1/4 of the tree. The following simple heuristic takes advantage of the above observation.

We precompute two vectors, \( \text{Left} \) and \( \text{Right} \), of length \( C - 1 \) where \( C \) is the number of columns in the data. The entries in the vectors are defined as follows: \( \text{Left}(i) \) is the first character in columns \( i \) and \( i + 1 \) where the columns differ when scanning from the left; \( \text{Right}(i) \) is the index of the first character that differs when scanning from the right. For instance, for columns \( s_5 = \text{AGAGAG} \), \( s_6 = \text{AGGAGAG} \) in the sorted version of our example data set (indexed 6 and 2, respectively, in Figure 1 before column sorting), \( \text{Left}(5) = 3 \), \( \text{Right}(5) = 7 \). The first mismatch scanning from the left is the A-G difference at position 3, and the first difference when scanning from the opposite end is the A-G found at position 7.

If the order of the columns is fixed, then computing vectors \( \text{Left} \) and \( \text{Right} \) simply requires comparing their entries in each column, and this calculation only has to be done once per data set. It is clear that the pass of the pruning algorithm for column \( i + 1 \) only needs to look at the leaves between \( \text{Left}(i) \) and \( \text{Right}(i) \), and at the internal nodes whose descendants include one of those leaves. Implementing this reduction is a matter of few simple changes to the modified pruning algorithm (see Algorithm 4 in the Appendix).

**RESULTS**

**Speed Gains**

We have tested the modified pruning algorithm on nucleotide, amino-acid, and codon data sets of various dimensions. The files used for testing may be downloaded from http://www.hyphy.org/pubs/cs/data.tar.gz, and results are found in Table 1.

There are two key observations in Table 1. First, the time reduction brought about by column sorting ranges from a minimum of near 10% to a maximum of over 80%, reflecting a fivefold improvement in speed. Trees with many sequences seem to benefit the most from the
TABLE 1. Effects of column sorting on computational speed. The columns are as follows: **Type**: Type of the data, i.e., nucleotide, amino acid, or codon. This determines N, the number of character states: 4 for nucleotides, 20 for amino acids, and 61 or 60 for codons (universal or mammalian mitochondrial genetic code). **Sites**: The number of distinct data columns (sites) in the data, i.e., the number of vertices in the graph G. Note that in real data this value is a function of the sequence length and the rates of evolution in different lineages. **Seqs**: The number M of sequences in the data file, which is the same as the number of leaves in the phylogenetic tree. **Time**: How long it took to carry out column sorting (in seconds). **Ref**: Number of likelihood evaluations per second, not using column sorting, but using the tree traversal heuristic. Calculations were done on a PowerMac G4/533 using HY-PHY version 0.95beta for MacOS X. Likelihood calculations also involve computing transition probabilities, and thus reflect a real-world speed gain from column sorting. **Sort**: Number of likelihood evaluations per second using both the column sorting algorithm and the tree traversal heuristic. **Speedup**: The ratio between the sorted and reference speeds. **BEP**: Break-even point. The number of likelihood evaluations per branch required to recover the overhead of column sorting. For example, the first entry requires 0.35 s to perform the column sorting. In that same 0.35 s, 0.35 x 293 likelihood evaluations could be performed. In order to allow comparison over trees with different numbers of taxa, we scale by the number of branches. BEP = 0.35 x 293/27 = 3.80.

<table>
<thead>
<tr>
<th>Type (N)</th>
<th>Sites</th>
<th>Seqs</th>
<th>Time</th>
<th>Ref</th>
<th>Sort</th>
<th>Speedup</th>
<th>BEP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuc (4)</td>
<td>725</td>
<td>15</td>
<td>.35</td>
<td>293</td>
<td>423</td>
<td>1.44</td>
<td>3.80</td>
</tr>
<tr>
<td>Nuc (4)</td>
<td>906</td>
<td>50</td>
<td>1.75</td>
<td>85</td>
<td>160</td>
<td>1.88</td>
<td>1.53</td>
</tr>
<tr>
<td>Nuc (4)</td>
<td>483</td>
<td>349</td>
<td>6.4</td>
<td>12.5</td>
<td>64</td>
<td>5.12</td>
<td>0.12</td>
</tr>
<tr>
<td>Nuc (4)</td>
<td>1264</td>
<td>500</td>
<td>77.15</td>
<td>3.3</td>
<td>8.1</td>
<td>2.46</td>
<td>0.26</td>
</tr>
<tr>
<td>Aa (19)</td>
<td>1519</td>
<td>37</td>
<td>0.05</td>
<td>42.1</td>
<td>72.9</td>
<td>1.73</td>
<td>0.05</td>
</tr>
<tr>
<td>Aa (20)</td>
<td>98</td>
<td>6</td>
<td>&lt;0.01</td>
<td>331</td>
<td>363</td>
<td>1.10</td>
<td>0.37</td>
</tr>
<tr>
<td>Cod (60)</td>
<td>1716</td>
<td>7</td>
<td>2.35</td>
<td>3.5</td>
<td>4.59</td>
<td>1.31</td>
<td>0.75</td>
</tr>
<tr>
<td>Cod (61)</td>
<td>513</td>
<td>23</td>
<td>0.67</td>
<td>2.55</td>
<td>3.9</td>
<td>1.53</td>
<td>0.04</td>
</tr>
</tbody>
</table>

**Quality of the Algorithm**

An ideal algorithm for likelihood calculations would reuse already computed partial likelihoods whenever possible, leading to no recalculations at all. The algorithm outlined in this paper only reuses partial likelihoods from the single previous step, and relies on an approximate solution to the TSP to sort columns. One can easily imagine modifying the sorting algorithm to store the past two sites, for instance, or even to store the partial likelihoods for all previous sites. However, such modifications would require additional memory and bookkeeping. It is desirable, though, to explore how much of the available savings our current method is able to exploit. In Table 3 we examine how well this approach compares with two possible algorithmic improvements.
This method has similar memory requirements to the TLB approach, but a simpler preprocessing step and poorer efficiency. It is included here only for comparative purposes, as the TLB method will always provide better improvements with a similar memory requirement.

Column sorting based on the minimal spanning tree.— Yet another idea would be to take better advantage of the minimum spanning tree. The Hamiltonian path approach is very memory efficient, but the resulting path can be up to twice the total length of the MST. In order to compute the likelihood of the entire data set, we can traverse the MST any way we choose, evaluating likelihoods of sites as we encounter them in the tree. One possibility is the one that led to the suboptimal Hamiltonian path. If, on the other hand, we traverse the tree pre-order, and at any given time store partial likelihoods for every site (internal node) on the path from the current node to the root of the tree (provided that the internal node has more than one child—see an example below), then the total cost of likelihood evaluations in terms of the metric defined in Equation 1 is simply the total length of the MST. The additional memory requirements will depend on the depth of the MST, i.e., the longest path between a leaf and the root. This quantity will depend on the phylogenetic tree and the alignment (which together determine the MST) and can be minimized by selecting the root of the MST appropriately. In example applications the memory requirement remains fairly small, compared to the FC (storing partial likelihoods for every site).

Consider an example MST shown in Figure 4, where the labels of every node mark the index of a corresponding hypothetical alignment site. The order of site computations and memory storage operations using this MST would be as follows (each computation except for first reuses all possible partial likelihoods from the site represented by the parent node in the MST): 13(store), 7 (store), 6 (store/flush 7), 5 (flush 6), 11 (store), 10, 12 (store/flush 11), 8, 9 (flush 12), 4 (store/flush 13), 3, 2, 1. At any given time, the partial likelihoods are stored for at most 3 sites.

The values in Table 2 represent the cost of likelihood evaluations in terms of the metric of this paper relative to the TLB (defined to be 1.0). It is clear that all of the alternative algorithms described in this section offer improved calculation times, but the gains tend to be relatively small. In time-critical settings, the additional memory and programming complexity of these algorithms may be justified, but it is pleasing to see that the simple one-step column sorting method captures much of the available savings.

Tree Traversal Savings Heuristic

Table 3 shows by how much the tree traversal heuristic reduces the number of nodes traversed by the pruning algorithm for each tree likelihood evaluation. Note that this heuristic doesn’t require that the columns be ordered in any particular way, but the sorting method of this paper boosts the efficiency of tree traversals quite a bit. The examples in Table 3 illustrate this observation. When applied to unsorted data sets, typical improvements in terms of branches traversed are in the vicinity of 5%. When this heuristic is applied to the sorted data, however, the improvement jumps to around 25%. The net improvement in terms of likelihood calculation speed is, predictably, rather small, but the heuristic can garner improvements of around 5% with essentially no computational overhead. Similar savings would be expected if the heuristic was used in conjunction with either the TLB or FC algorithms. Like column sorting, the use of the heuristic requires little overhead and only minor changes to the pruning algorithm, so it is recommended for typical likelihood implementations.

**DISCUSSION**

A few points regarding evolutionary models should be made. First of all, in the version of the algorithm described here, all sites are assumed to have equal evolutionary rates. When site-to-site heterogeneity is introduced, the form of the likelihood function is slightly modified (Yang, 1994), requiring computation of a series of conditional likelihoods for each likelihood evaluation. The column sorting approach applies directly to this situation. Other models, such as the spacial rate correlation model of Felsenstein and Churchill (1996) are also

**TABLE 3. Tree traversal heuristic savings.** Table entries indicate the relative reduction in the number of branches traversed when using the tree traversal heuristic to compute likelihoods of all data columns. **Natural** and **Sorted** refer to the ordering of data columns, and each of those columns contains the percent reduction in the number of traversed branches as a result of using the traversal heuristic. **Speedup** reflects relative improvement of likelihood evaluations per second, when the tree traversal heuristic is applied to sorted data columns. Data sets are again numbered as in Table 1.

<table>
<thead>
<tr>
<th>Data set</th>
<th>Natural</th>
<th>Sorted</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.60</td>
<td>23.82</td>
<td>1.04</td>
</tr>
<tr>
<td>2</td>
<td>8.46</td>
<td>28.92</td>
<td>1.07</td>
</tr>
<tr>
<td>3</td>
<td>8.47</td>
<td>35.45</td>
<td>1.08</td>
</tr>
<tr>
<td>4</td>
<td>4.19</td>
<td>13.64</td>
<td>Negligible</td>
</tr>
<tr>
<td>5</td>
<td>3.88</td>
<td>35.48</td>
<td>1.05</td>
</tr>
<tr>
<td>6</td>
<td>2.27</td>
<td>40.14</td>
<td>1.06</td>
</tr>
<tr>
<td>7</td>
<td>1.26</td>
<td>43.37</td>
<td>Negligible</td>
</tr>
<tr>
<td>8</td>
<td>0.65</td>
<td>33.14</td>
<td>Negligible</td>
</tr>
</tbody>
</table>

**Figure 4.** Example of a MST.
suitable for column sorting. A second important point is that the column sorting algorithm can be fine-tuned for specific evolutionary models, taking into account asymmetries of the substitution probabilities. For instance, the two columns (AAAAGG) and (CCCCCT) would require recalculation of all partial likelihoods using the current algorithm. However, if we use the Jukes and Cantor (1969) model, there is no need for recalculation. Similar savings can be found for other models.

The column sorting algorithm described here offers speed improvements in settings where the likelihood function is evaluated multiple times for the same phylogenetic tree. Typical likelihood-based analyses involving phylogenetic trees require iterative optimization of the likelihood function and entail repeated evaluation of the likelihood for different model parameters, but leave the tree and the data unchanged. Thus, one column sorting step may be used to trim off computational costs in hundreds or even thousands of likelihood function evaluations. Although the algorithm is not guaranteed to provide a benefit for every data set, it is easy to implement and has little computational or memory overhead, and can provide substantial time improvements for many data sets.

ACKNOWLEDGMENTS

The work of SVM was supported by NSF grant DBI-0096033, and SLKP was supported by an NSF IGERT Fellowship. We thank John Huelsenbeck and an anonymous reviewer for useful comments on the manuscript. We are indebted to John Kececioglu for his suggestions on additional uses of the MST.

REFERENCES


First submitted 24 October 2002; reviews returned 21 April 2003; final acceptance 5 April 2004
Associate Editor: John Huelsenbeck

APPENDIX

Algorithm 1: Distance Computation
If we assume that sites i and j differ in at least one position, one way to compute $d(s_i, s_j)$ defined in equation (1) is as follows:

```plaintext
// Leaf labels are characters from the appropriate column of the alignment
distance := 0;
treeNode := first node in post-order traversal;
WHILE (treeNode is not the root) DO
  IF treeNode is a leaf THEN
    IF leaf label from site i is different from label at site j THEN
      mark parent of treeNode as tainted
    END IF
  ELSE
    IF treeNode is marked as tainted THEN
      distance := distance + number of children of treeNode
    END IF
    mark parent of treeNode as tainted
    treeNode := next node in post-order traversal
  END IF
END WHILE
distance := distance + number of children of the root
```

Algorithm 2: Approximate Solution of the TSP
When the following algorithms refer to vertices and edges, those are the components of the distance graph (as described in “Reduction to a Graph Traversal Problem,” and not the phylogenetic tree.

Algorithm 2a: Constructing a suboptimal Hamiltonian path.—The algorithm proceeds in two stages:

1. Find a MST $M$ of the graph $G$ (e.g., using Prim’s algorithm; see below).
2. Conduct a pre-order traversal of $M$, outputting the vertices of the tree in the order they were traversed.

The resulting ordering of vertices is the desired suboptimal path.

Algorithm 2b: Constructing a MST.—Prim’s algorithm for constructing the MST is described in detail in Gibbons (1985). The outline of the algorithm is as follows:

1. Choose the longest edge (breaking ties arbitrarily) in $G$ and add one of its vertices to the set $V$.
2. Repeat the following three steps until all vertices of the graph are in $V$:
   (a) Choose the shortest edge $e$ between a vertex in the set $V$ and a vertex not in $V$ (again, breaking ties arbitrarily).
   (b) Add the edge $e$ to the MST.
   (c) Add the vertex upon which $e$ is incident to $V$.

Algorithm 3: Modified Pruning Algorithm
The following pseudocode illustrates the modifications to Felsenstein’s pruning algorithm required by column sorting.

```plaintext
// Assume that the columns have already been sorted
// Leaf labels are characters from the appropriate column of the alignment
```
Use the standard pruning algorithm to compute all partial likelihoods for column $i=1$.

FOR $i:= 2$ to (number of columns) DO
  treeNode := first node in post-order traversal
  WHILE (treeNode is not the root) DO
    IF treeNode is a leaf THEN
      IF leaf label from site $i$ is different from label at site $i - 1$ THEN
        relabel treeNode with the appropriate sequence character
        mark parent of treeNode as tainted
      END IF
    ELSE
      IF treeNode is marked tainted THEN
        compute the vector of partial likelihoods at treeNode
        mark parent of treeNode as tainted
        unmark treeNode
      END IF
    END IF
    lastNode := treeNode
    treeNode := next node in post-order traversal
  END WHILE
  compute tree likelihood by weighting partial likelihoods at the root by equilibrium frequencies
  update likelihood for the entire data set
END FOR

**Algorithm 4: Tree Traversal Cost Reduction Algorithm**

// Change the body of the FOR loop in the modified pruning algorithm as follows

treeNode := Leaf indexed by $[\text{Left}(i-1)]$
WHILE (treeNode is not the root AND treeNode is not equal to Leaf indexed by $[\text{Right}(i-1)+1]$) DO
  IF treeNode is a leaf THEN
    IF leaf label from site $i$ is different from label at site $i - 1$ THEN
      relabel treeNode with the appropriate sequence character
      mark parent of treeNode as tainted
    END IF
  ELSE
    IF treeNode is marked tainted THEN
      compute the vector of partial likelihoods at treeNode
      mark parent of treeNode as tainted
      unmark treeNode
    END IF
  END IF
  lastNode := treeNode
  treeNode := next node in post-order traversal
END WHILE
  compute tree likelihood by weighting partial likelihoods at the root by equilibrium frequencies
  update likelihood for the entire data set