FastTree: Neighbor-Joining with Profiles instead of a Distance Matrix

Morgan N Price*[1,2] , Paramvir S Dehal[1,2]and Adam P Arkin[1,2,3]

[1]Physical Biosciences Division, Lawrence Berkeley National Lab, 1 Cyclotron Road Mailstop 977-152, Berkeley California 94720, USA
[2]Virtual Institute of Microbial Stress and Survival, Lawrence Berkeley National Lab, Berkeley California, USA
[3]Department of Bioengineering, University of California, Berkeley, California, USA

Email: Morgan N Price*- morgannprice@yahoo.com;

*Corresponding author

Abstract

Background: A fundamental goal of molecular evolution is to infer the evolutionary history – the phylogeny – of sequences from their alignment. Neighbor joining, which is the standard method for inferring large phylogenies, takes as its input the distances between all pairs of sequences. The distance matrix requires O(N^2L) time to compute and O(N^2) memory to store, where N is the number of sequences and L is the width of the alignment. As some families already contain over 100,000 sequences, these time and space requirements are prohibitive.

Results: We show that neighbor-joining can be implemented in O(NLa) space, where a is the size of the alphabet, by storing profiles of internal nodes in the tree instead of storing a distance matrix. Profile-based neighbor joining allows weighted joins, as in BIONJ, but requires that distances be linear. With heuristic search, neighbor joining with profiles takes only O($N\sqrt{N}\log(N)La$) time. We estimate the confidence of each split (A,B) vs. (C,D) from the profiles of A, B, C, and D, without bootstrapping. Our implementation, FastTree, has similar accuracy as traditional neighbor joining. FastTree constructed trees, including support values, for biological alignments with 39,092 or 158,022 distinct sequences in less time than it takes to compute the distance matrix and in a fraction of the space. Traditional neighbor joining with 100 bootstraps would be 10,000 times slower.

Conclusions: Neighbor joining with profiles makes it possible to construct phylogenies for the largest sequence families and to estimate their reliability. FastTree is available at http://microbesonline.org/fasttree.

Background

Inferring phylogenies from biological sequences is the fundamental method in molecular evolution, and has many applications in taxonomy and for predicting structure and biological function. In general, sequences are identified as homologous, aligned, and then a phylogeny is inferred. Large alignments can be constructed efficiently, in O(NL^2) time, by aligning the sequences to a profile instead of to each other (e.g., with hmmalign from the HMMer package [1]).

Neighbor joining [2–4] is the most popular method for constructing large phylogenies. Neighbor joining is also often used to generate an initial tree before searching for the maximum likelihood tree (e.g., in PhyML [5]). Neighbor joining relies on a "distance matrix" that stores an estimate of the evo-

lutionary distance between each pair of sequences. Estimating distances requires comparing the entries at each position in the alignment and hence requires $O(L)$ time per entry and $O(N^2L)$ time overall, and the distance matrix requires $O(N^2)$ space to store. The neighbor-joining method itself requires a further $O(N^3)$ time, but there are heuristic variants that take only $O(N^2)$ or $O(N^2 \log N)$ time without significant additional memory overhead and with little loss of accuracy [6, 7].

As DNA sequencing accelerates, the memory and CPU requirements of the distance matrix approach are becoming prohibitive. Many families already contain 100,000-200,000 members: for example, the MicrobesOnline database [8], which provides phylogenies for all protein families from prokaryotic genomes [9], already contains 100 protein families that contain over 100,000 distinct sequences. Similarly, an alignment of full-length 16S ribosomal RNAs from greengenes [10,11] contains over 160,000 distinct sequences. The distance matrix for families with 100,000-200,000 members requires 20-80 gigabytes (GB) of memory to store (a 4-byte floating point value for each of $N(N-1)/2$ pairs). Although computers with this much memory are available, the typical node in a compute cluster has an order of magnitude less memory. Furthermore, DNA sequencing technology is improving rapidly, and the distance matrix's size scales as the square of the family's size, so we expect these problems to become much more severe.

If an estimate of the reliability of the tree is desired, then the usual approach is the bootstrap [12], which requires rerunning the method 100-1,000 times. The increase of 100-fold means that the computation can take weeks of CPU time for a single family.

Results and Discussion

Our Approach

We present FastTree, which uses three ideas to reduce the space and time complexity of inferring a phylogeny from an alignment (Figure 1). First, FastTree stores profiles for the internal nodes in the tree instead of storing a distance matrix. Although methods for inferring phylogenetic trees are usually classified as being either distance-based or character-based, FastTree is both: FastTree builds profiles at the internal nodes, and yet it gives exactly the same result as neighbor joining (if the alignment does not contain gaps). Each profile includes a frequency vector for each position, and the profile of an internal node is the weighted average of its childrens' profiles. FastTree uses these profiles to compute the distances between internal nodes. These profiles require a total of $O(NLa)$ space, where a is the size of the alphabet (20 for protein sequences and 4 for nucleotide sequences), instead of $O(N^2)$ space for the distance matrix. However, the time required for neighbor-joining with exhaustive search rises from $O(N^3)$ to $O(N^3La)$, because every distance has to be recomputed on demand in $O(La)$ time.

Second, FastTree uses a combination of previously published heuristics [6, 7] and a new "top hits" heuristic to reduce the number of joins considered. Whereas traditional neighbor joining considers $O(N^3)$ possible joins, and previous heuristics have considered $O(N^2)$ possible joins (the size of the distance matrix), FastTree considers $O(N\sqrt{N}\log N)$ possible joins. Thus, in theory, FastTree takes $O(N\sqrt{N}\log(N)La)$ time. In practice, FastTree is faster than computing the distance matrix.

Third, FastTree computes "local" support values for internal nodes by examining the profiles. Examining the profiles around every internal split takes $O(NLa)$ time, and thus has negligible cost relative to inferring a tree. This gives FastTree an additional 100-fold speed-up relative to neighbor-joining with bootstrap.

A major limitation of FastTree is that it only supports linear distance measures. By linear we mean that the distance betwen two sequences i and j is an average over positions:

$$d(i,j) = \frac{\sum_{l=1}^{L} D(i_l, j_l)}{L}$$

where l are the positions in the alignment and D is a distance measure on nucleotides or amino acids. Linear distances are not an accurate reflection of evolution: they do not correct for hidden changes on long branches (e.g. if A mutates to C and then back to A) and they saturate (the maximum possible distance between two sequences is 1 substitution per site). Correcting for the hidden changes, by using log-corrected distances [13] or maximum-likelihood distances [14], leads to a modest improvement in tree accuracy [15]. In theory, maximum-likelihood distances should be more accurate than log-corrected

distances, but in practice, both methods lead to trees of similar accuracy [15], and log-corrected distances are about 1,000 times faster to compute (data not shown). We will show that despite its use of linear distances, FastTree is about as accurate as traditional neighbor joining with maximum-likelihood or log-corrected distances.

Traditional Neighbor-joining

Given a distance matrix, neighbor joining is a greedy algorithm for finding a tree of (nearly) minimal length [2–4]. It begins with a star topology, in which every sequence is a child of the root, and it joins pairs of nodes until it the tree has only three children. At each step, it selects the join that will yield the largest reduction in total tree-length by finding the pair of nodes that minimize the "neighbor-joining criterion"

$$d'(i,j) \equiv d(i,j) - r(i) - r(j)$$

$$r(i) \equiv \frac{\sum_{k \neq i} d(i,k)}{n-2}$$

where i, j, k are indices of active nodes that have not yet been joined, $d(i,j)$ is the distance between nodes i and j, n is the number of active nodes, $r(i)$ can be thought of as the average "out-distance" of i to other active nodes (although the denominator is $n-2$, not $n-1$), and $d'(i,j)$ is the criterion.

Once it selects a pair of nodes i, j to join, neighbor joining creates a new internal node ij that is the parent of i and j, and sets i and j as inactive. It sets distances for the new node according to the "reduction" rule:

$$d(ij,k) = \frac{d(i,k) + d(j,k) - d(i,j)}{2}$$

and the distance from the new node to its children is given by

$$d(ij,i) = \frac{d(i,j) + r(i) - r(j)}{2}$$

and similarly for $d(ij,j)$.

Neighbor-joining with Profiles

FastTree computes the neighbor-joining criterion $d'(i,j)$ in $O(La)$ time without storing a distance matrix. Instead, FastTree stores the aligned input sequences and the profiles of the interior nodes. At each position, the profile of a new joined node ij is the average of the profiles of i and j. For example, if we join two leaves i and j, and i has an A at a position and j has a G, then the profile of ij at that position will be 50% A and 50% G (and 0% for other characters). If we then join ij to kl, where kl's profile is 40% A and 60% T, then the new node $ijkl$ will have the profile 45% A, 25% G, and 30% T at that position.

If the distances are linear, then the average of $d(i,k)$ and $d(j,k)$, which is used to compute $d(ij,k)$, can be computed by comparing the profile of k to the profile for ij. The remaining term in $d(ij,k)$, the $-d(i,j)/2$ term, can be thought of as a correction for the distance up from i or j to ij. We store this "up-distance" $u(ij)$ when we create the joined node. For leaves, no correction is needed, and the "up-distance" is zero. Thus, for any pair of nodes i and j, we write

$$d(i,j) = P(i,j) - u(i) - u(j)$$

where $P(i,j)$ is a "profile distance" that is the average distance between profile characters over all positions:

$$P(i,j) \equiv \frac{\sum_{l=1}^{L} D(\vec{P}_l(i), \vec{P}_l(j))}{L}$$

$$D(\vec{P}_l(i), \vec{P}_l(j)) \equiv \sum_{a,b} P_l(i,a) P_l(j,b) D(a,b)$$

where $P_l(i,a)$ is the frequency of a in the profile of i at position l and $D(a,b)$ is the distance measure on characters. For example, if we use a simple %different notion of distance, so that the distance between two characters is 0 if they match or 1 otherwise, then the distance between frequency vectors $(0.6, 0.4, 0, 0)$ and $(0.5, 0, 0.5, 0)$ is $0.6 \cdot 0.5 \cdot 0 + 0.6 \cdot 0.5 \cdot 1 + 0.4 \cdot 0.5 \cdot 1 + 0.4 \cdot 0.5 \cdot 1 = 0.7$.

The total distance to other nodes, which is required to compute the $r(i)$ term in the criterion, can be computed by comparing the profile of i to a "total profile" which is the average over all active nodes:

$$\sum_{j=1}^{n} P(i,j) = nP(i,T)$$

where T is the total profile. Thus, computing the criterion $d'(i,j)$ requires three profile comparisons

(for $P(i,j)$, $P(i,T)$, and $P(j,T)$). The distance between two profiles can be computed in $O(La)$ time, instead of the naive $O(La^2)$ time, by using the eigen decomposition of the alphabet's distance matrix (see Methods). Thus, FastTree takes $O(La)$ time to compute the neighbor-joining criterion. In the Methods section, we give a formula for $u(ij)$ and we show that, with this formula, FastTree gives the exact same values as traditional neighbor joining for the distance between internal nodes and for the neighbor joining criterion if the alignment contains no gaps and the distance between sequences is linear.

With ungapped sequences and a linear distance measure, profiles can also be used to give an exact implementation of weighted neighbor-joining methods such as BIONJ [16]. BIONJ achieves a modest increase in accuracy [15] by weighting long branches (rapidly evolving sequences) lower when combining $d(i,k)$ and $d(j,k)$ to give $d(ij,k)$. With profiles, this is represented by weighting the contribution of i and j to the profile of ij and by modifying the formula for the up-distances $u(ij)$. In traditional BIONJ, the weight for the join is computed from a "variance matrix" which is obtained from the distances by another reduction formula. As this reduction formula is also linear, it can also be implemented by profiles (see Methods).

Gaps

The description in the previous section assumes that the alignment has no gaps. Indeed, alignment columns that contain gaps are often removed, which is known as "trimming" the alignment. Gapped columns are removed both because they often represent uncertain positions in the alignment and because, even for correct alignments, it is not clear how gaps should be interpreted. However, for large sequence families, trimming away all of the gaps is not practical: very few positions are represented in every member of the sequence, and trimming away every position that contains a gap would eliminate most of the phylogenetic signal. Instead, the gaps are usually treated as missing data. When computing the distance between two sequences, it is straightforward to remove positions that are missing from either sequence.

If the alignment does contain gaps, then FastTree will not give the exact same result as traditional neighbor joining. FastTree records the fraction of gaps at each profile position, and when computing distances, FastTree weights positions by their proportion of non-gaps. If $d(A,B) = d_{AB}/n_{AB}$, where n_{AB} is the number of positions that are not gaps in either sequence, and $d(B,C) = d_{BC}/n_{BC}$ then the profile method will approximate the average of the two distances by $(d_{AB} + d_{BC})/(n_{AB} + n_{BC})$. For example, consider this alignment of three sequences and seven positions with the alphabet {0,1}:

```
1010-10 (A)
10111-0 (B)
1011-11 (C)
101?..0 (AB) profile
```

where the profile for `AB` is shown with "." for part-gap positions and "?" for a mixture of 0 and 1. $d(A,C) = 2/6$ and $d(B,C) = 1/5$. By the profile approach, $P(AB,C) = (1+0.5)/(5+0.5) = 3/11 = 0.273$, but $(d(A,B) + d(A,C))/2 = (2/6 + 1/5)/2 = 0.267$. As this example illustrates, the change due to using profiles is not large. A similar issue also arises with estimating the average distance to other nodes $r(i)$ (formulas for $r(i)$ in the presence of gaps are given in the Methods). We will show that this approximation leads to acceptable results in practice, even for highly gapped alignments.

Heuristics for neighbor joining in $O(N^2 \log(N) La)$ time

Traditional neighbor joining conducts an exhaustive search for the best join at each step. There are $N-3$ joins, and $n(n-1)/2$ candidate joins at each step, where n is the number of active nodes, so it takes $O(N^3)$ time to search for the best joins. This dominates the running time. We use a combination of two previously published heuristics, FastNJ [6] and relaxed neighbor joining [7], to reduce the number of joins considered at each step from $O(n^2)$ to $O(n \log n)$. In the next section, we will show how to further reduce the number of joins considered.

The key idea in FastNJ is to store the best join for each node. The best join for each leaf is determined before the joins begin, and the best join for each new interior node is determined when that node is created. When searching for the best join overall, FastNJ considers only best join for each node, or n candidates. Thus, FastNJ requires a total of $O(N^2)$ time. Although FastNJ gives correct results on distance matrices that closely approximate distances on a tree, it suffers from a small reduction in accuracy in practice [6]. The best join for a node according to the neighbor-joining criterion can change over time

because the $r(i)$ values change after every join.

In relaxed neighbor joining, a local hill-climbing search is used to find a "locally best" join. Given an (arbitrary) node A, it will find the best join partner B for A, and then the best join partner C for B. If A=C, then A and B are each other's best hit and it has reached a local optimum; otherwise it continues searching from (B,C). To avoid very poor local optima, relaxed neighbor joining also adds a check to ensure that is not lengthening the tree. (If it is, it starts over with another node.) Relaxed neighbor joining takes $O(N^2 \log N)$ time and is reported to have about the same accuracy as traditional neighbor joining [7], although it results in a small loss of accuracy in our tests (see below).

To combine these two heuristics, we store the best hit for each node and we search for the best join among these n best hits, as in FastNJ. Then, we do a local hill-climbing search, as in relaxed neighbor joining. Because we are always starting with a "relatively good" join, we omit the check on the total tree length. Finally, if we create a new interior node and find that it is a better join for an existing node than the previously known best hit for that node, we update that node's best hit. This greatly reduces the amount of hill-climbing search that is required (data not shown).

In theory, this method takes $O(N^2 \log NLa)$ time, because hill-climbing takes an average of $O(\log N)$ iterations to converge, it is run on each of $N-3$ joins, and each hill-climbing step computes the neighbor joining criterion $O(N)$ times. However, hill-climbing is rarely necessary: for example, when we ran FastTree on 39,092 distinct sequences from Pfam PF00005, it took only 433 hill-climbing steps, or ≈ 0.01 steps per join, rather than $\log_2 N = 15.3$ steps per join. Thus, the running time seems to be $O(N^2La)$ in practice.

Top-hits heuristic for neighbor joining in $O(N\sqrt{N}\log(N)La)$ time

To reduce the search time further, we use a top-hits heuristic. The intuition is that if A and B are close to each other, then the top hits of A and B will largely overlap. Similarly, when we join A and B, we expect that the best hit of the new node AB will be found among the top hits of A and B. We store the top $m = \sqrt{N}$ hits for each node.

To initialize the top-hit lists, before we do any joins, we take an arbitrary "seed" sequence, we compare it to all other sequences by using the neighbor joining criterion, and we select the top m results. This takes $O(NL)$ time. The top m hits of the seed are likely to be close neighbors. (FastTree has additional checks to verify that the top hit lists are likely to be similar – see Methods.) We can approximate the top hit list of each neighbor by comparing that neighbor to the top $2m$ hits of the seed, where the two is an arbitrary safety factor. This takes $O(m^2L)$ time. We can then remove these close neighbors from the list of seed sequences. This method considers up to N/m seeds and hence takes a total of $O(N^2L/m + NmL)$ time. Because we use $m = \sqrt{N}$, this reduces to $O(N\sqrt{N}L)$ time, which is the best possible with this approach.

When we join A and B, we merge the top hit lists to get up to $2(m-1)$ candidates, we compare the new node to those candidates, and we store the top m hits. If the top hit lists of A and B are identical, then we will be left with only $m-1$ top hits for AB (we must remove A and B). Thus, we allow the top hit list to drop below m in size (m does not change while the algorithm runs).

Also notice that when we join A and B, any hits to A or B in the top-hit lists of other nodes should be replaced by hits to AB. FastTree does this in a "lazy" way: when it encounters a hit to a joined node, it replaces that with a hit to the active ancestor. If another node has top hits to both A and B, then both hits would eventually be replaced by AB, so this is another way that the top-hit lists become shorter.

If the top-hit list has shrunk too much (below $0.8m$, where 0.8 is an arbitrary parameter), then we "refresh" – we recompute the top-hit list for the new joined node and we update the top-hit lists of the new node's top hits. FastTree also refreshes if this part of the tree has not been refreshed in many joins (see Methods). To refresh, we compare the new node to all $n-1$ other active nodes, and we save the top m hits. Then, we compare the close neighbors of AB (the top m hits) to the top $2m$ hits of AB, and we update the close neighbors' top-hit lists by merging. Refreshing a top-hit list takes $O(nLa + m^2La)$ $= O(NLa)$ time and ensures that the top-hit lists of $O(m = \sqrt{N})$ other nodes reach size m. Thus, the refreshes take a total of $O(N\sqrt{N}La)$ time. We also note that if the top hit list has size equal to the remaining number of active nodes (e.g. because $n \leq m$), then the top-hit lists are exhaustive. At that point there is only $O(m^2La = NLa)$ work re-

maining.

Finally, given the top-hit lists, we need to find the best join in $O(mLa)$ time. We use the heuristics described in the previous section, with the further restriction that we consider only the top m candidates at each step. Thus, we first find the best m joins among the best-hit entries for the n active nodes. In principle, this can be implemented in $O(m \log n)$ time by using a priority queue. (FastTree simply sorts all n entries, which adds a total of $O(N^2 \log N)$ to the running time. For 100,000 sequences, this is not significant compared to the $O(N\sqrt{N} \log(N)La)$ time for the other steps.) Then, we compute the current value of the neighbor-joining criterion for those m candidates, which takes $O(mLa)$ time, and we select the best one. Given this candidate join, we do a local hill-climbing search for a better join, but we only search within the top-hit lists rather than comparing the two nodes to all other active nodes, so this takes $O(m \log(N)La)$ time.

When we do a join, we also need to update the total profile (the average over all active nodes). To compute this profile takes $O(nLa)$ time. However, we can subtract the joined nodes and add the new node to the total profile in $O(La)$ time. (FastTree recomputes the total profile from scratch every 200 iterations to avoid roundoff errors from accumulating, where the choice of 200 is arbitrary. This adds another $O(N^2La/200)$ work.)

In theory, this method takes $O(N\sqrt{N}L)$ time to find the top hits for the sequences, $O(N\sqrt{N} \log(N)La)$ time to search for the best joins, and $O(N\sqrt{N}La)$ time to update the top-hit lists, for a total of $O(N\sqrt{N} \log(N)La)$ time. Because some sequences may not have $O(m)$ close neighbors, it is not clear if our method can truly attain $O(N\sqrt{N} \log(N)La)$ time without sacrificing accuracy. Nevertheless, we will show that for large problems, FastTree is faster than computing the distance matrix. We will also show that the top-hit heuristic does not reduce accuracy.

"Local" Support Values

Whatever the method used, inferred phylogenies often contain errors, and so it is essential to estimate the reliability of the result [17]. The standard method to estimate reliability is to use the bootstrap [12]: to resample the columns of the alignment, to rerun the method 100-1,000 times, to compare the resulting trees to each other or to the tree inferred from the full alignment, and to count the number of times that each split occurs in the resulting trees. (A split is the two sets of leaves on either side of an internal edge.) Bootstrap estimates reliability, not accuracy: the inferred tree may be incorrect because of bias (e.g., long branch attraction), and not just because of limited phylogenetic signal (e.g., not enough positions in the alignment). Thus, splits that are found in all or almost all of the resampled trees may yet be incorrect. Nevertheless, the bootstrap is by far the most popular method to check the reliability of phylogenetic inference.

Unfortunately, bootstrapping is a minimum of 100 times slower than the underlying phylogenetic inference. Commonly used tools, such as phylip's consense [14] and quicktree [18], also use $O(N^3)$ time and $O(N^2)$ space to compare two trees to each other (data not shown). Tree comparison could be implemented in $O(N^2)$ time and $O(N)$ space by using a hash table to look up the lists of nodes on each side of a split. (The lists of nodes are represented by the trees and need not be stored.) In any case, because of these memory and time requirements, bootstrapping a tree with just 10,000 sequences takes days of CPU time with current tools (see below).

To estimate the reliability of the tree more quickly, we compute a local support value using a minimum evolution framework (that is, the assumption that the tree with the shortest total branch-length is the best tree). As neighbor joining is a greedy search for a minimum evolution tree [4], this is the appropriate framework to use to test the reliability of a neighbor joining tree. The concept of local support was originally introduced in a maximum likelihood framework [19]. Our method also seems similar to the minimum-evolution test [20], although as far as we know that test has not been implemented efficiently for large numbers of sequences.

We measure how strongly the minimum evolution criterion supports each split over the other two possible topologies around that edge. This is a local support value because we do not consider how this change might alter other parts of the tree (or the profiles for the 4 nodes around this edge). More precisely, we estimate the probability P that support this strong would occur by chance, and we report $1 - P$ as a support value. Alternatively, FastTree also includes the option to compute a local bootstrap (the proportion of resampled alignments that support this split over the two alternate splits).

Given four subtrees A,B,C,D, the tree

((A,B),(C,D)) is preferred by the minimum evolution criterion if

$$d(A,B) + d(C,D) < \min(d(A,C) + d(B,D), \\ d(A,D) + d(B,C))$$

where the two terms in the minimum correspond to the alternate topologies ((A,C),(B,D)) and ((A,D),(B,C)). This comparison can be done using profile-distances, without considering the "up-distances", because changing all distances for a node by a constant does not affect the relative values of the above criterion.

We compute the amount by which ((A,B),(C,D)) is favored over each of the other two topologies at each position, which gives two sets of values

$$x_{1l} \equiv P_l(A,C) + P_l(B,D) - P_l(A,B) - P_l(C,D)$$
$$x_{2l} \equiv P_l(A,D) + P_l(B,C) - P_l(A,B) - P_l(C,D)$$

over positions l, where $P_l(A,B)$ is the profile distance between A and B at that position. Given linear distances, the minimum evolution criterion can be written as

$$\min(\bar{x}_1, \bar{x}_2) > 0$$

where $\bar{x}_1$ is the mean of $\{x_{1l}\}$. To compute a local bootstrap, we can resample x_1 and x_2 over positions and determine how often this minimum evolution criterion is met.

We can also use x_1, x_2 to ask how often such strong support would occur by chance under the null hypothesis that the true phylogeny is a star topology (A,B,C,D). We treat the two sets as correlated normal variables with means of zero (as implied by the null hypothesis of a star topology). We test how often a value as high as the observed $y = \min(\bar{x}_1, \bar{x}_2)$ would be expected by chance. There are three possible splits, for which neighbor joining usually selects the best one, so the under the null hypothesis, the cumulative distribution of y is given by

$$P(y) = P(\min(\bar{x}_1, \bar{x}_2) > y | \min(\bar{x}_1, \bar{x}_2) > 0) \\ = 3 \cdot P(\min(\bar{x}_1, \bar{x}_2) > y)$$

The distribution of $\min(\bar{x}_1, \bar{x}_2)$ is roughly normal because the two variances are similar and the correlation is nonnegative (data not shown). So, we estimate the variances of x_1 and x_2 and their correlation from the data, and we use the analytic formulas for the mean and variance of the minimum of two normals [21] to describe the distribution of $\min(\bar{x}_1, \bar{x}_2)$. We then use a one-tailed normal test to estimate $P(\min(\bar{x}_1, \bar{x}_2) > y)$ and the formula above to give $P(y)$ and hence the support value $1 - P(y)$. We will show that in practice, these support values are effective at distinguishing correct splits from incorrect splits.

To compute x_1, x_2 we need additional profiles beyond those constructed during neighbor joining. For example, to test the split of DE versus ABC in the tree (A,B,(C,(D,E))) we need profiles for AB as well as for C, D, and E, but the only internal profiles constructed during neighbor joining are for (D,E) and for (C,(D,E)). These additional profiles can be computed by depth-first search from the root in $O(NLa)$ time and take a maximum of $O(dLa)$ space to store, where d is the maximum number of edges down to any leaf from the root. (There are $O(N)$ additional profiles but only $O(d)$ of them are required at any one time.) In practice, $d \ll N$, so computing the support values costs negligible time or space relative to constructing the tree.

Performance of FastTree

Topological Accuracy in Simulations

To verify that profile-based neighbor joining returns the same results as traditional neighbor joining for ungapped alignments, we simulated 150 ungapped alignments of 40 protein sequences with realistic phylogenies and alignment widths by using Rose [22] (see Methods for details). For each alignment, we ran FastTree with exhaustive search and we ran BIONJ, a weighted variant of neighor joining [16], with the corresponding distance matrix. We used linear distances with distances between individual amino acids derived from the BLOSUM45 [23] amino acid similarity matrix (see Methods). For all of these alignments, FastTree and BIONJ gave the same topology and the same branch lengths (within round-off error). If we used unweighted joins instead of weighted joins, FastTree gave the same results as quicktree, an implementation of traditional neighbor joining [18].

As explained above, profile-based neighbor joining cannot give the same result as traditional neighbor joining on alignments with gaps. FastTree's approximation may be particularly inaccurate when the number of gaps varies across sequences (see Methods). To create alignments with an extreme variation in the number of gaps, we truncated the sequences in these simulations at one end, so that

they ranged from having no gaps to being 80% gaps. We measured the topological accuracy of the resulting trees (that is, the proportion of splits in the true trees that were recovered by FastTree or by traditional neighbor joining). The topological accuracy of FastTree (with weighted joins and again with exhaustive search) was virtually identical to that of BIONJ (both were 63.2%), which suggests that FastTree's handling of gaps is acceptable.

To test whether the search heuristics in FastTree give acceptable accuracy, we simulated 81 gapped alignments of 2,000 protein sequences with realistic phylogenies and alignment widths. We also tested several other implementations of neighbor joining, all using the same distance measure derived from BLOSUM45 [23] (see Methods). We found that FastTree was 4.1% less accurate than BIONJ and 1.6% less accurate than quicktree, but FastTree was 3.0% more accurate than clearcut, an implementation of relaxed neighbor-joining [7] (see Table 1). All of these differences were statistically significant (all $P < 10^{-15}$, paired t test, $n = 81$). The loss of accuracy due to the best hits, local hill-climbing, and top-hits heuristics was negligible: heuristic FastTree was about 0.1% less accurate than FastTree with exhaustive search, and this difference was not statistically significant ($P > 0.3$). The weighted joins in FastTree did give a slight (0.6%) but statistically significant improvement in accuracy ($P < 10^{-12}$). We are not sure why FastTree is less accurate than BIONJ – because FastTree with exhaustive search gives the same results as BIONJ for non-gapped alignments, it may be due to FastTree's approximations for gaps.

One limitation of FastTree relative to the distance matrix methods is that FastTree cannnot use non-linear distances. Log-corrected distances gave a small increase in accuracy for both BIONJ and clearcut (Table 1). However, clearcut was still less accurate than FastTree (68.4% vs. 70.9%, $P < 10^{-15}$). As reported previously [15], BIONJ gave trees of about the same accuracy with either log-corrected or maximum-likelihood distances (only 0.12% different, see Table 1).

Although FastTree is less accurate than some other neighbor-joining methods, it is not clear if this is significant in practice. Most of the erroneous splits found by FastTree have poor support (see below). Conversely, we found that most splits missed by FastTree but found by BIONJ have poor support. To quantify the support, we used the approximate likelihood ratio test (aLRT) of PhyML [19, 24] (see Methods for details). For neighbor joining trees, aLRT is a better measure of support than the bootstrap (see below). We considered a split to have significant support if the aLRT value was 90% or higher. We found that BIONJ with log-corrected distances, the best method in Table 1, inferred 54.2% of splits correctly and with significant support, while FastTree with default settings (heuristic search and weighted joins) inferred 53.8% of splits correctly and with significant support. This 0.4% difference in accuracy for well-supported splits is much smaller than the 4.7% overall difference. More broadly, maximum likelihood methods are more accurate than neighbor joining, but most of those additional correct splits also have poor support [17]. Thus, the additional correct splits found by methods other than FastTree generally have poor support and might not be useful for drawing biological conclusions.

Quality of Trees for Genuine Protein Families

To test the quality of FastTree's results on genuine protein families, we inferred topologies with FastTree and with other implementations of neighbor joining and we used the tree's likelihood (after optimizing the branch lengths with PhyML [5]) as an estimate of the tree's quality. As shown in Table 2, for alignments of 500 sequences, FastTree gives results of the same quality as BIONJ with either log-corrected distances or maximum likelihood distances, and FastTree gives significantly better results than either clearcut or quicktree. Surprisingly, FastTree with heuristic search gives slightly better results than FastTree with exhaustive search. This effect is statistically significant, but it is small, and it is not entirely consistent: exhaustive search returns trees with higher likelihoods for 99 of the 304 families. Overall, these tests confirmed that the quality of FastTree's trees is competitive with those from traditional neighbor joining.

Quality of Support Values

To compare our support values to the traditional bootstrap, we simulated gapped alignments of sequences by using Rose [22]. We simulated 150 ungapped alignments of 40 protein sequences each with realistic topologies and alignment widths, and we inferred a topology from each alignment with FastTree. The inferred topologies were 76.6% accurate. We

computed local (probabilistic) support values and local bootstrap with FastTree, as described above, and we computed traditional bootstrap by resampling each alignment 100 times with SEQBOOT [14], running FastTree on each resample, and recording how often each split appeared in the resampled trees.

The local support values were highly correlated with traditional bootstrap values (Spearman rank correlation $\rho = 0.89$). Local bootstrap had a similar correlation with traditional bootstrap ($\rho = 0.88$). To quantify the ability of each measure to distinguish correct splits from incorrect splits, we used the Kolmogorov-Smirnov D-statistic, a non-parametric measure of the dissimilarity of two distributions that ranges from 0 for identical distributions to 1 for non-overlapping distributions. All three measures had a similar ability to distinguish correct splits ($D = 0.65$ for local support, $D = 0.66$ for local bootstrap, and $D = 0.64$ for traditional bootstrap).

We also compared FastTree's support values to a maximum-likelihood implementation of local support, the approximate likelihood ratio test [19] of PhyML [5]. FastTree's local support values were strongly correlated with support values from PhyML aLRT ($\rho = 0.87$), but PhyML aLRT was better at distinguishing correct splits than either FastTree or traditional bootstrap ($D = 0.73$ versus $D = 0.65$ or $D = 0.64$).

Finally, for incorrect splits, the distribution of the local support values should be uniform. As shown in Figure 2C, the distribution of local support values is roughly uniform in practice. There is a peak near zero, which reflects highly uncertain splits, but this does not affect the interpretation of significant values (0.95 or above). For incorrect splits, FastTree's support values are more uniform than the traditional bootstrap or the local bootstrap. The figure also illustrates that FastTree's support values between 0.8 and 0.9 should not be interpreted as offering much support, as is sometimes done with bootstrap values (only 75% of such splits were correct).

Computational Performance

To test FastTree's performance on genuine families, we ran FastTree on a gene family from the COG database [25] (COG2814), a domain family from PFam [26] (PF00005), and a trimmed alignment of all sequenced full-length 16S rRNAs [10,11]. These families contain 8,362, 39,092, and 158,022 distinct sequences, respectively. As shown in Table 3, FastTree is faster than computing the distance matrix, and for PF00005 and 16S rRNA it requires far less memory. If estimates of the tree's reliability is required, then even for COG2814, FastTree is 1,000 times faster than traditional neighbor joining (quicktree) with 100 bootstraps. Maximum-likelihood methods (e.g. PhyML [5] or RAxML [27]) were even slower than quicktree with bootstrap.

For the 16S alignment, the only other method that seems practical is clearcut. Clearcut itself is very fast – we estimate that it might take only 10 hours to infer a tree from the 16S distance matrix. However, clearcut requires a distance matrix, and FastTree is faster than clearcut once the cost of computing the distance matrix is included. Clearcut would also require over 50 gigabytes of memory – 20 times as much as FastTree – which makes it impractical for us to run. Unlike FastTree, Clearcut does not produce support values – producing those would take at least 100 times longer (or over half a year). Furthermore, clearcut seems to be less accurate than FastTree (Tables 1 & 2). Overall, we found that FastTree scales to the largest sequence families, while the distance matrix methods have prohibitive CPU and memory requirements.

Future Work

Log-corrected distances

Log-corrected distances of the form $-\log(1-d)$ yield a small improvement in accuracy (Tables 1 & 2; [15]), presumably because they correct for back-mutations on long branches and thus help to avoid long-branch attraction. FastTree cannot use log-corrected distances because there is no way to compute the average of the $\log(1-d)$ values from the profiles. However, it might be possible to achieve the same effect by log-correcting the distances between nodes $d(i,j)$ and the "out-distances" $r(i)$. A more sophisticated distance measure could also be used to compute support values.

Branch-swapping

After neighbor joining is completed, FastTree already tests whether the topology around internal nodes are consistent with the minimum evolution criterion. In a significant minority of cases (14% of splits for PF00005), the minimum evolution criterion prefers an alternate topology, and FastTree

could change the topology (a nearest-neighbor interchange). In our simulations, the majority of these "bad splits" are incorrect (data not shown), so altering them should improve accuracy. The nearest-neighbor interchanges might take as little as $O(N \log(N) La)$ time, so they need not increase the running time of FastTree too much. It should also be possible to use a higher-quality distance metric on profiles during this step.

Nearest-neighbor interchanges with a minimum evolution criterion have previously been implemented in a distance matrix approach in FastME [28]. Although FastME gives better results than unweighted neighbor joining, it seems to be less accurate than BIONJ [15]. We suspect that this is because FastME uses unweighted joins, and that nearest-neighbor interchanges with weighted joins would improve accuracy. In any case, it should be possible to improve the accuracy of FastTree to that of traditional neighbor joining with high-quality distances.

Progressive Multiple Sequence Alignment

In this work, and in our web site tools (MicrobesOnline [9]), we rely on profile-based multiple sequence alignment as the most practical method for the largest families. However, profile-based alignment is believed to be less accurate than progressive alignment. One of the limiting steps in progressive alignment is the construction of the guide tree. The top-hit heuristic, or perhaps FastTree itself, might be useful for this step. For example, PartTree [29] uses a divide-and-conquer algorithm and k-mer distances to compute a (lower accuracy) guide tree on unaligned sequences in $\mathrm{O}(N \log(N) L)$ time. This tree could be used to generate an initial alignment, and FastTree could be used to generate the guide tree for another iteration of progressive alignment.

In principle this approach could lead to accurate progressive alignments in less than $\mathrm{O}(N^2 L)$ time. However, the number of gaps in an alignment grows with the number of sequences, because of independent insertions in the subfamilies. To achieve progressive alignment in less than $\mathrm{O}(N^2 L')$ time, where L' is the length of the longest input sequence, would require masking out subfamily-specific insertion positions while analyzing the other parts of the tree.

Conclusions

FastTree implements neighbor joining with profiles of internal nodes instead of an explicit distance matrix. The main limitations of FastTree are that distances must be linear and that approximations are required to handle gaps. However, with both simulated and genuine protein alignments, FastTree is about as accurate as traditional neighbor-joining methods that use a distance matrix. On simulated data, FastTree was slightly less accurate than the best of these methods [15], BIONJ with log-corrected or maximum-likelihood distances, but on genuine families, FastTree and BIONJ gave trees of the same quality (as measured by likelihood). In both tests, FastTree more accurate than either unweighted neighbor joining with %different distances (quicktree) or relaxed neighbor joining (clearcut).

FastTree uses three heuristics to reduce the search effort for finding the best join: it stores the best-hit of each node [6], it uses local hill-climbing [7], and it estimates the top hits a node from the top hits of a close neighbor. These heuristics do not lead to any appreciable loss of accuracy, and because of them, FastTree requires less than $O(N^2 L)$ time in theory. In practice, for the largest alignments, FastTree is faster than computing the distance matrix. Thus, FastTree is faster than any method that requires a distance matrix, and much faster than traditional neighbor joining ($\mathrm{O}(N^3)$ time).

FastTree computes local support values rather than using the bootstrap, which gives FastTree another 100-fold speed-up. For families of under 10,000 sequences, this is probably the main practical advantage of FastTree. FastTree's local support values have a roughly uniform distribution for the incorrectly inferred splits and are about as effective as traditional bootstrap in distinguishing correct splits from incorrect splits. However, the scale of the support values may be different. Whereas bootstrap proportions as low as 0.8 are often interpreted as significant support, in our experience, only FastTree support values of 0.9 or above indicate that a split is probably correct. But because FastTree's support values estimate the strength of rejection of a star topology, for difficult problems where most splits are unresolvable, the posterior probability of a split being correct, given a support value of 0.9, could be lower.

Finally, because FastTree stores $\mathrm{O}(N)$ profiles of size $\mathrm{O}(La)$ each instead of an $O(N^2)$ distance matrix, FastTree requires far less memory than tradi-

tional neighbor joining. As families grow larger, this will make distance matrix approaches untenable: for the 16S alignment, with 158,022 distinct sequences, the distance matrix is already 50 GB. Thus, FastTree makes it practical to infer phylogenies for families with hundreds of thousands of sequences. These phylogenies will be useful for reconstructing the tree of life and for predicting functions for the millions of uncharacterized proteins that are being identified by large-scale DNA sequencing.

Methods

Details of FastTree

Exact distances between internal nodes for gap-free alignments

Here we show that, for gap-free alignments, we can compute the distances between internal nodes, according to the reduction formula of neighbor-joining or BIONJ, from the profiles and the "up-distances" $u(i)$. When we join two nodes i and j, we store a profile or a frequency vector at each position l for the new parent node ij as the (weighted) average

$$\vec{P}_l(ij) = \lambda\vec{P}_l(i) + (1-\lambda)\vec{P}_l(j)$$

where λ is the weight for BIONJ or $\lambda = 1/2$ for neighbor joining. Because the profile distances $P(i,j)$ are linear,

$$P(ij,k) = \lambda P(i,k) + (1-\lambda)P(j,k)$$

Now, if we first consider unweighted joins, the reduction formula for neighbor joining is [3]

$$d(ij,k) = \frac{d(i,k)+d(j,k)-d(i,j)}{2}$$

and we write

$$d(ij,k) = P(ij,k) - u(ij) - u(k)$$

$$u(ij) = \frac{P(i,j)}{2}$$

and $u(l) = 0$ for leaves. For two leaves i and j, $P(i,j) = d(i,j)$, so this gives the correct distance between leaves. Assume the distances are correct for all nodes so far and consider the next join ij:

$$d(ij,k) = \frac{d(i,k)+d(j,k)-d(i,j)}{2}$$

$$\begin{aligned}
&= \frac{1}{2}(P(i,k) - u(i) - u(k) \\
&\quad + P(j,k) - u(j) - u(k) \\
&\quad - P(i,j) + u(i) + u(j)) \\
&= \frac{P(i,k)+P(j,k)}{2} - \frac{P(i,j)}{2} - u(k) \\
&= P(ij,k) - u(ij) - u(k)
\end{aligned}$$

which shows that our distances are correct for $d(ij,k)$ and, by induction, for all nodes. For weighted joins, a similar argument shows that

$$u(ij) = \lambda(u(i) + d(i,ij)) + (1-\lambda)(u(j) + d(j,ij))$$

gives the same result as the distance reduction formula for BIONJ [16]

$$\begin{aligned}
d(ij,k) &= \lambda(d(i,k) - d(i,ij)) \\
&\quad + (1-\lambda)(d(j,k) - d(j,ij))
\end{aligned}$$

For BIONJ, we also need to reduce the "variance" matrix. The variance values for pairs of leaves are the same as the distance values, and the BIONJ reduction formula for variances is [16]

$$v(ij,k) = \lambda v(i,k) + (1-\lambda)v(j,k) - \lambda(1-\lambda)v(i,j)$$

which can be computed from profile-distances by using a "variance correction" $\nu(i)$ analogous to the up-distances, where $\nu(i) = 0$ for leaves and

$$v(i,j) = P(i,j) - \nu(i) - \nu(j)$$

$$\nu(ij) = \lambda\nu(i) + (1-\lambda)\nu(j) + \lambda(1-\lambda)v(i,j)$$

Given these variances, BIONJ weights the join of i,j so as to minimize the variance of the distance estimates for the new node ij, using the formula

$$\lambda = \frac{1}{2} + \frac{\sum_{k\neq i,j}(v(j,k) - v(i,k))}{2(n-2)v(i,j)}$$

where n is the number of active nodes before the join takes place.

Handling gaps

Here we explain how, in the presence of gaps, FastTree can still achieve a good approxmation of neigh-

bor joining. The distance between two profiles becomes

$$P(i,j) \equiv \frac{\sum_{l=1}^{L} D(\vec{P}_l(i), \vec{P}_l(j)) w_l(i) w_l(j)}{\sum_{l=1}^{L} w_l(i) w_l(j)}$$

where $w_l(i)$ is the proportion of non-gaps for i at position l. The proportion of non-gaps for an internal node is just the weighted average of the values for its children. (The profiles' frequency vectors do not include gaps.) As explained above, these profile distances are good but not exact estimates of the weighted averages of distances between nodes.

The key complication introduced by gaps is the computation of the out-distances $r(i)$. In the absence of gaps, the average profile distance between a node and all other nodes can be inferred from the total profile T:

$$r(i) = \frac{\sum_{j \neq i} d(i,j)}{n-2}$$

$$\sum_{j \neq i} d(i,j) = \sum_{j \neq i} (P(i,j) - u(i) - u(j))$$

$$= \sum_{j} P(i,j) - P(i,i) - (n-1)u(i) - \sum_{j \neq i} u(j)$$

$$= nP(i,T) - P(i,i) - (n-1)u(i) - (\sum_{j} u(j) - u(i))$$

which can be computed in $O(La)$ time if we store the total profile T and the total of all the up-distances.

In the presence of gaps, this formula is not a good approximation because highly gapped sequences contribute little to the total profile. Instead, we need to take the weights of the comparisons into account. Let $T - i$ be the total profile with the contribution from i removed. Then

$$\sum_{i \neq j} P(i,j) \approx (n-1) P(i, T-i)$$

$$P(i, T-i) = \frac{\sum_{j \neq i} \sum_{l=1}^{L} P_l(i,j) w_l(i) w_l(j)}{\sum_{j \neq i} \sum_{l=1}^{L} w_l(i) w_l(j)}$$

$$P(i,T) = \frac{\sum_{j} \sum_{l=1}^{L} P_l(i,j) w_l(i) w_l(j)}{\sum_{j} \sum_{l=1}^{L} w_l(i) w_l(j)}$$

which leads to a formula for $P(i, T-i)$ in terms of $P(i,T)$ and $P(i,i)$ and the total weights of those comparisons. In practice, this gives a good approximation for the out-distances in the presence of gaps (data not shown).

However, this introduces a small bias into the neighbor joining criterion. An intuitive justification of the neighbor joining criterion is that it is not altered if we increase the rate of evolution along one branch, or, in other words, we increase all $d(i,j)$ for a given i by the same amount. This is why neighbor joining correctly estimates trees rather than clustering the sequences. For simplicity, we will multiply the neighbor joining criterion by $n-2$ to get

$$(n-2)d(i,j) - \sum_{k \neq i} d(i,k) - \sum_{k \neq j} d(j,k)$$

If we increase $d(i,k)$ by 1.0 for all k then this criterion for i,j changes by

$$(n-2) - (n-1) - 1 = -2$$

and the criterion for j,k changes by

$$0 - 1 - 1 = -2$$

which is this same, which shows that neighbor joining is not affected by long branches if the distances are correct. Now imagine that i is half gaps but all other sequences are ungapped. Then i will contribute less than other sequences to the out-distances of other nodes. Again, increase all $d(i,k)$ by 1.0. Then our criterion for i,j changes by

$$(n-2) \cdot 1.0 - 1.0 \cdot (n-1) - 0.5 = -1.5$$

and our criterion for j,k changes by

$$0 - 0.5 - 0.5 = -1.0$$

so in this case, the i,j join is (incorrectly) preferred.

Thus, FastTree seems to be biased towards joins that involve long-branched sequences if they have

many gaps, or towards joins that involve short-branched sequences if they have few gaps. We suspect that this is the reason for the loss of accuracy of FastTree relative to BIONJ on the large simulated alignments, even though we detected no loss of accuracy on smaller highly-gapped alignments.

Finally, gaps also complicate the interpretation of the "variances" used for weighted joins. In principle, the variances should be divided by the number of non-gap positions in the comparison, as distances that are computed from more positions are more reliable. However, if we do that, the reduction formula for variances given in the previous section becomes unreliable (data not shown). Instead, we implicitly weight less-gapped sequences more highly because the less-gapped member of a join contributes more strongly to the profile.

Top-hit Heuristics

FastTree searches within the top $2m$ hits of a seed sequence to estimate the top m hits for a "close neighbor." A close neighbor is defined as a sequence that does not yet have a top-hits list and is one of the top m hits of the seed. To ensure that these neighbors are "close enough" so that the top-hits heuristic is likely to be accurate, FastTree also requires that the distance between the seed and the close neighbor be at most 75% of the distance to the seed's $2m$th-best hit. (Here we use actual distances, not the neighbor joining criterion.) For ungapped sequences the top-hit list is guaranteed to be correct if the neighbor is less than half of the distance to the worst hit considered, because of the triangle inequality. On the other hand, for a perfectly balanced tree with clock-like evolution, we expect the distance of hit m to be proportionate to $\log_2 m$ and the distance of hit $2m$ to proportionate to $1 + \log_2 m$. Thus, 75% is an intermediate between a conservative value and one that gives O(m) close neighbors per leaf under ideal conditions. FastTree has an option (-close) to change this parameter for faster performance.

FastTree also has a requirement that the close neighbor have a similar pattern of gaps as the seed. This is to avoid cases where the sequences overlap in only a few positions, so that they might be identical or nearly so (for the positions considered) even though they have very different top-hits lists. Specifically, the requirement is that the total number of non-gap positions in the seed/neighbor comparison must be at least $1 - d/2$ times the number of non-gap positions in the neighbor, where d is the maximum distance allowed for a close neighbor, or at least $1 - 2d/3$ times the average shared positions between the seed and its top $2m$ hits.

FastTree also has a heuristic to refresh the top-hit lists when it has been "too long" since the last refresh. (This is in addition to refreshing top-hits lists that are too short.) To do this we store an "age" for each node. The age is zero for the leaves and for refreshed nodes. For each join, we set $age(ij) = 1 + \max(age(i), age(j))$. We refresh when the age reaches $\log_2 m$, so that we expect to do a refresh for every 1 out of m joins, or $O(m)$ refreshes overall.

Finally, given the top-hit heuristic as described so far, the top-hit lists eliminate much of the node-node comparisons, and a significant fraction of the remaining work is in recomputing the out-distances $r(i)$. When the number of active nodes is high, a single join has little effect on the total profile or on the out-distances, so this work is not useful. We use "stale" values of the out-distances to compute the neighbor joining criterion as long as the number of active nodes has changed by less than 2% since the out-distance for that node was last computed. (The choice of 2% is arbitrary.)

Distances between amino acids

The simplest estimate of the evolutionary distance between two aligned sequences is the fraction of non-gap positions that differ. Indeed, by default, FastTree uses this metric for nucleotide alignments. However, some substitutions are more likely than others, and for protein sequences, accounting for this leads to better estimates of evolutionary distance and hence to more accurate trees (Table 2).

We set the distance between two amino acids to be proportionate to the negative log of their similarity. By default, FastTree uses the BLOSUM45 similarity matrix [23], but in principle, any similarity matrix that is designed for distantly related sequences would be appropriate. We normalize the distances between amino acids so that the expectation of the distance for unrelated sequences is 1. (More precisely, we compute the average of each row or column in the distance matrix, weighted by the frequency of the amino acids, and we divide each entry by the harmonic mean of the row and column averages.) Alternatively, you can specify your own amino acid distance matrix.

To compute the distance between two profile positions in $O(a)$ instead of $O(a^2)$ time, we use the eigen decomposition of the amino acid distance matrix D. Rather than storing the profiles $\vec{P}_l(i)$, we actually store rotated profiles $\vec{Q}_l(i)$:

$$D = V^{-1}\Lambda V$$

$$\vec{Q}_l(i) = V^{-1}\vec{P}_l(i)$$

$$\begin{aligned} D(\vec{P}_l(i), \vec{P}_l(j)) &= \sum_{a,b} P_l(i,a)P_l(j,b)D(a,b) \\ &= \vec{P}_l(i)D\vec{P}_l(j) = (V^{-1}\vec{P}_l(i))\Lambda(V^{-1}\vec{P}_l(j)) \\ &= \vec{Q}_l(i)\Lambda\vec{Q}_l(j) = \sum_a Q_l(i,a)Q_l(j,a)\Lambda_a \end{aligned}$$

This computation takes $O(a)$ time but is still much slower than computing the distance between two characters, which takes only a matrix lookup. So, we record which positions match a single character (i.e., the frequency vector contains a single value of 1.0 and the other values are 0) and we avoid comparing the frequency vectors if both positions are single characters. This also gives significant memory savings because we do not need to store the profile for that position (we can use the rotated vector for that character instead).

Unique sequences

Large alignments often contain many sequences that are exactly identical to each other [18]. FastTree uses hashing to quickly identify redundant sequences, constructs a tree for the unique subset of sequences, and then creates multifurcating nodes, without support values, as parents of the redundant sequences.

Alignments

Data Sources

We obtained members of COG gene families [25] and of Pfam [26] PF00005 from the fall 2007 release of the MicrobesOnline database [9]. In MicrobesOnline, the sequences are aligned to the family's profile rather than by progressive alignment – the COG alignments are from reverse position-specific blast [30] and the PF00005 alignment is from hmmalign [1]. As the profiles only include positions that are present in many members of the family, these alignments do not contain all positions from the original sequences. The 16S rRNA alignment is from greengenes [10,11] and is trimmed according to the greengenes mask.

Simulated families

For simulations with 2,000 sequences, we used the 81 COG alignments that had over 2,000 distinct sequences. For simulations with 500 and 40 sequences we also used COGs that exceeded the required size. Given an actual family and alignment, we selected the desired number of family members (at random and after removing duplicate sequences) and we inferred a phylogeny using traditional neighbor-joining (quicktree). For the 40-sequence simulations we computed distances with protdist from the phylip package [14]; for the 2,000-sequence simulations we used log-corrected distances with the BLOSUM45 matrix. The average total length of the tree was 29.6 substitutions per site for 40 sequences and 530.0 substitutions per site for 2,000 sequences.

From these realistic phylogenies, we simulated protein sequence evolution with Rose [22], using an insertion and deletion rate of 0.0005 (or 0 for ungapped simulations), the default settings (the PAM matrix), and 16 categories of gamma-distributed rates with a shape parameter of 0.5. We removed positions from the alignment that were $\geq 90\%$ gaps ($\geq 95\%$ for simulations with 40 members). Note that we used the true (simulated) alignment rather computing an alignment from the simulated sequences.

Comparisons to Other Methods

Distances for Traditional Neighbor Joining

Our log-corrected distance uses the formula $-1.3\log(1 - d(i,j))$, which is similar to but simpler than scoredist's [13] formula

$$-1.33 \cdot \log \frac{s(i,j) - s_0}{\frac{s(i,i)+s(j,j)}{2} - s_0}$$

where $s(i,j)$ is the similarity score and s_0 is the expected value of the similarity score for two random non-homologous sequences. The two formulas for log-corrected distances give similar results because our distance measure on amino acids is defined so that $1 - d(i,j)$ is approximately proportionate to $s(i,j) - s_0$ and because the denominator in the scoredist formula is roughly constant. As in scoredist, we

also truncate our log-corrected distances to values no greater than 3.0, and for sequences that do not overlap because of gaps, we use this maximum distance.

We implemented our own log-corrected distances, rather than using the implementation of scoredist in Belvu, because of Belvu's excessive CPU and memory requirements. However, in smaller simulations, BIONJ trees based on either log-corrected distances or scoredist distances were equally accurate (data not shown).

We computed maximum likelihood distances with protdist 3.65 [14] and default settings (JTT matrix, one category of rates). Because protdist occasionally gives unreasonable values (e.g., distances of -1 for non-overlapping sequences, or values as high as 50 for distantly related sequences), we set a maximum distance of 3, and we replaced values below 0 with 3.

PhyML aLRT Support Values

To compute support values for a tree with PhyML aLRT, we optimized the branch lengths and we used the combined supports (option -3), which are a minimum of support values from the parametric test [19] and from a non-parametric variant [24]. We used the JTT model, no invariant sites, and no rate variation across sites.

CPU time and memory

Performance was measured on a computer with 4 dual-core 2.6 GHz AMD Opteron processors and 32 GB of RAM. All programs used a single thread of execution. Programs were downloaded from the authors' web sites or were compiled with gcc version 3.4.6 and the -O2 optimization setting. The versions of software tested were FastTree 0.9, quicktree 1.1, clearcut 1.0.8, PhyML aLRT 1.1 (64-bit executable), a C implementation of BIONJ [31], and RAxML VI version 1.0 [27]. For RAxML, which is another maximum likelihood method, we used the BIONJ tree as a starting tree, and the fast hill-climbing option (-f d). For best performance, PhyML and RAxML were run with no variation of rates across sites. Neither job finished: PhyML crashed on COG2814 after the likelihood had nearly converged, and we are not sure why the RAxML job stopped running.

The time to compute the distance matrix (in Table 3) was measured by running FastTree with the -makematrix option, so that it outputs a distance matrix in phylip format rather than building a tree. To save memory, it computes the distances between all N^2 pairs of sequences.

To estimate the performance of the distance matrix methods on the larger families (in Table 3), we extrapolated from COG2814 and we used the scaling behavior of each algorithm. We assumed that BIONJ scales as $O(N^3)$ time and $O(N^2)$ space and that clearcut scales as $O(N^2 \log N)$ time and $O(N^2)$ space. The scaling of quicktree is cubic in time and quadratic in space, but we scaled by the number of distinct sequences, as quicktree includes an optimization to remove redundant sequences (as does FastTree).

Authors contributions

M.N.P. developed, implemented, and tested the algorithm. All authors analyzed the data and wrote the manuscript.

Acknowledgements

This work was supported by a grant from the US Department of Energy Genomics:GTL program (DE-AC02-05CH11231).

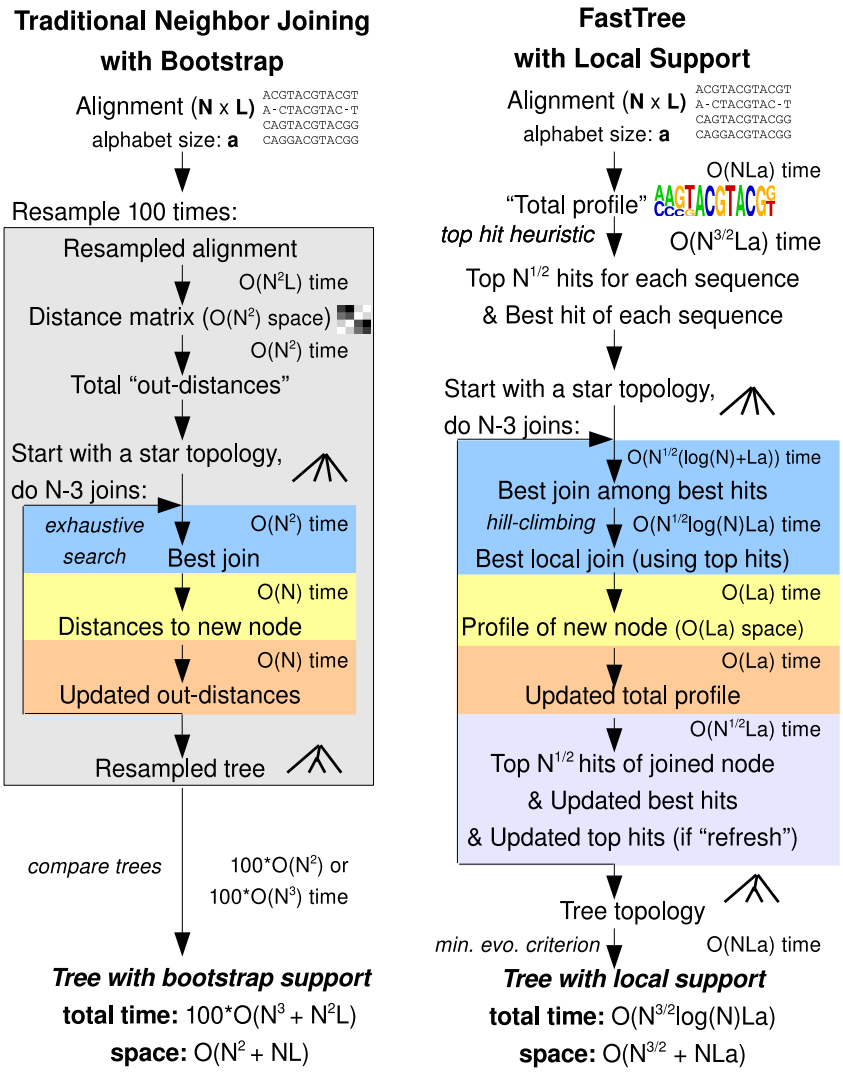
References

1. **HMMER: biosequence analysis using hidden markov models**[http://hmmer.janelia.org/].
2. Saitou N, Nei M: **The neighbor-joining method: a new method for reconstructing phylogenetic trees**. *Mol. Biol. Evol.* 1987, **4**:406–425.
3. Studier JA, Keppler KJ: **A note on the neighbor-joining algorithm of Saitou and Nei.** *Mol Biol Evol.* 1988, **5**:729–31.
4. Gascuel O, Steel M: **Neighbor-Joining Revealed**. *Mol Biol Evol.* 2006, **23**:1997–2000.
5. Guindon S, Gascuel O: **A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood**. *Syst Biol.* 2003, **52**:696–704.
6. Elias I, Lagergren J: **Fast Neighbor Joining**. In *Proc. of the 32nd International Colloquium on Automata, Languages and Programming (ICALP'05), Volume 3580 of* Lecture Notes in Computer Science, Springer-Verlag 2005:1263–1274.
7. Evans J, Sheneman L, Foster J: **Relaxed neighbor joining: a fast distance-based phylogenetic tree construction method.** *J Mol Evol.* 2006, **62**:785–92.

8. Alm EJ, Huang KH, Price MN, Koche RP, Keller K, Dubchak IL, Arkin AP: **The MicrobesOnline Web site for comparative genomics.** *Genome Res.* 2005, **15**:1015–22.
9. **How to Use the MicrobesOnline TreeBrowser**[http://www.microbesonline.org/treebrowseHelp.html].
10. DeSantis TZ, Hugenholtz P, Larsen N, Rojas M, Brodie EL, Keller K, Huber T, Dalevi D, Hu P, Andersen GL: **Greengenes, a Chimera-Checked 16S rRNA Gene Database and Workbench Compatible with ARB**. *Appl Environ Microbiol* 2006, **72**:5069–5072.
11. **Greengenes: aligned 16S rRNA data and tools**[http://greengenes.lbl.gov].
12. Felsenstein J: **Confidence Limits on Phylogenies: An Approach Using the Bootstrap**. *Evolution* 1985, **39**:783–791.
13. Sonnhammer ELL, Hollich V: **Scoredist: A simple and robust protein sequence distance estimator**. *BMC Bioinformatics* 2005, **6**:108.
14. **PHYLIP home page**[http://evolution.genetics.washington.edu/phylip.htm].
15. Hollich V, Milchert L, Arvestad L, Sonnhammer EL: **Assessment of protein distance measures and tree-building methods for phylogenetic tree reconstruction.** *Mol Biol Evol.* 2005, **22**:2257–64.
16. Gascuel O: **BIONJ: an improved version of the NJ algorithm based on a simple model of sequence data**. *Mol Biol Evol.* 1997, **14**:685–695.
17. Nei M, Kumar S, Takahashi K: **The optimization principle in phylogenetic analysis tends to give incorrect topologies when the number of nucleotides or amino acids used is small.** *Proc Natl Acad Sci USA* 1998, **95**:12390–7.
18. Howe K, Bateman A, Durbin R: **QuickTree: building huge Neighbour-Joining trees of protein sequences.** *Bioinformatics* 2002, **18**:1546–7.
19. Anisimova M, Gascuel O: **Approximate likelihood-ratio test for branches: A fast, accurate, and powerful alternative.** *Syst Biol.* 2006, **55**:539–52.
20. Rzhetsky A, Nei M: **Simple Method for Estimating and Testing Minimum-Evolution Trees**. *Mol Biol Evol.* 1992, **9**:945–967.
21. Kella O: **On the distribution of the maximum of bivariate normal random variables with general means and variances**. *Commun. Statist.-Theory Meth.* 1986, **15**:3265–3276.
22. Stoye J, Evers D, Meyer F: **Rose: generating sequence families**. *Bioinformatics* 1998, **14**:157–163.
23. Henikoff S, Henikoff JG: **Amino Acid Substitution Matrices from Protein Blocks**. *Proc. Natl. Acad. Sci. USA* 1992, **89**.
24. **PhyML aLRT**[http://atgc.lirmm.fr/alrt/].
25. Tatusov RL, Natale DA, Garkavtsev IV, Tatusova TA, Shankavaram UT, Rao BS, Kiryutin B, Galperin MY, Fedorova ND, Koonin EV: **The COG database: new developments in phylogenetic classification of proteins from complete genomes.** *Nucleic Acids Res.* 2001, **29**:22–8.
26. Finn RD, Mistry J, Schuster-Bckler B, Griffiths-Jones S, Hollich V, Lassmann T, Moxon S, Marshall M, Khanna A, Durbin R, Eddy SR, Sonnhammer EL, Bateman A: **Pfam: clans, web tools and services**. *Nucleic Acids Res.* 2006, **34**:D247–51.
27. Stamatakis A: **RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models**. *Bioinformatics* 2006, **22**:2688–2690.
28. Desper R, Gascuel O: **Theoretical Foundation of the Balanced Minimum Evolution Method of Phylogenetic Inference and Its Relationship to Weighted Least-Squares Tree Fitting**. *Mol. Biol. Evol.* 2004, **21**:587–598.
29. Katoh K, Toh H: **PartTree: an algorithm to build an approximate tree from a large number of unaligned sequences**. *Bioinformatics* 2007, **23**:372–374.
30. Schaffer AA, Aravind L, Madden TL, Shavirin S, Spouge JL, et al.: **Improving the accuracy of PSI-BLAST protein database searches with composition-based statistics and other refinements.** *Nucleic Acids Res.* 2001, **29**:2994–3005.
31. **BIONJ.c**[http://www.lirmm.fr/~w3ifa/MAAS/BIONJ/BIONJ.c].

Figures

Figure 1 - An overview of traditional neighbor joining and of FastTree.



Neighbor Joining Criterion: find the join that minimizes

$$d'(A,B) = d(A,B) - r(A) - r(B)$$

Distances to joined nodes:

Neighbor joining: $d(AB,C) = (d(A,C)+d(B,C))/2 - d(A,B)/2$

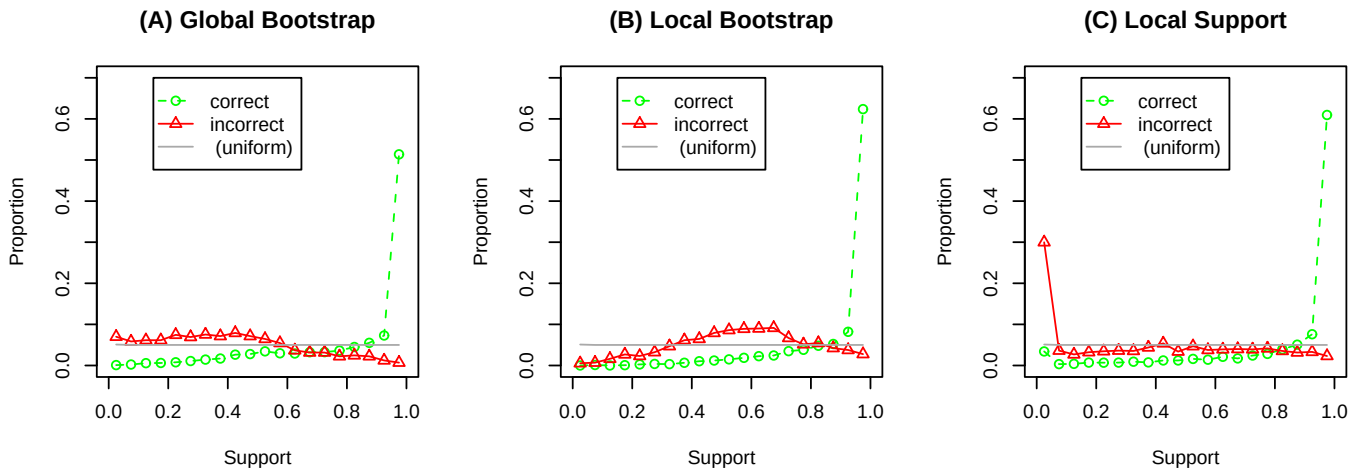
FastTree: $= P((A+B)/2,C) - u(C) - u(AB)$

Average out-distances:

Neighbor joining: $r(A) = \sum d(A,X)/(n-2)$

FastTree: $= \dfrac{n*P(A, \sum X/n) - P(A,A) - \sum (u(A)+u(X))}{n-2}$

Figure 2 - Distribution of support values for simulated alignments of 40 protein sequences with gaps. We compare the distribution of three different types of support values for correctly- and incorrectly-inferred splits: (A) the traditional (global) bootstrap, with 100 replicates; (B) FastTree's local bootstrap, with 100 replicates; and (C) FastTree's probabilistic local support. The horizontal line shows the uniform distribution.



Tables

Table 1 - Accuracy of various implementations of neighbor joining on simulated protein families with 2,000 sequences and realistic topologies.

We estimated topologies from 81 COG families [25] with over 2,000 members, we simulated gapped protein sequences with Rose [22], and we removed alignment positions that were $\geq$ 90% gaps (see Methods for details). We inferred trees with each method, using linear distances derived from the BLOSUM45 amino acid similarity matrix. We also ran BIONJ and clearcut with log-corrected distances (also based on BLOSUM45), BIONJ with maximum likelihood distances (from protdist [14]), and quicktree with %different distances. We report the average topological accuracy of each method, with the most accurate methods listed first. FastTree's default settings are highlighted in italics.

Program	Distances	Search	Joins	Accuracy
BIONJ	log-corrected	exhaustive	weighted	75.6%
BIONJ	max. likelihood	exhaustive	weighted	75.5%
BIONJ	linear	exhaustive	weighted	75.0%
quicktree	linear	exhaustive	unweighted	72.5%
FastTree	linear	exhaustive	weighted	71.0%
FastTree	*linear*	*top-hit*	*weighted*	*70.9%*
FastTree	linear	top-hit	unweighted	70.3%
clearcut	log-corrected	relaxed	unweighted	68.4%
clearcut	linear	relaxed	unweighted	67.8%
quicktree	%different	exhaustive	unweighted	65.5%

Table 2 - Quality of trees on genuine protein families of 500 sequences.

For each of the 304 COG families [25] that have over 500 members in MicrobesOnline [8], we randomly selected 500 distinct member sequences, we aligned them via their psi-blast profile [30], and we inferred phylogenies with a variety of methods. To estimate the quality of each tree's topology, we used the tree's log likelihood, as reported by PhyML [5] after optimizing branch lengths. To put these likelihoods on a comparable scale, we use the value relative to the best log likelihood for that alignment. We report this average loss in log likelihood for each method, with the best (least loss) methods listed first. We also show if the method is significantly worse than FastTree with the top-hit heuristic (by a two-tailed t test). The alignments ranged from 65 to 1,009 amino acids wide (mean 323) and contained 2-35% gaps (mean 10%).

Program	Distances	Search	Joins	Δ log lik.	P(t test)
FastTree	linear BLOSUM45	top-hit	weighted	-113.1	–
BIONJ	max. likelihood	exhaustive	weighted	-114.3	1
BIONJ	log BLOSUM45	exhaustive	weighted	-133.0	0.3
FastTree	linear BLOSUM45	exhaustive	weighted	-143.4	$9 \cdot 10^{-14}$
quicktree	log BLOSUM45	exhaustive	unweighted	-189.5	0.001
BIONJ	linear BLOSUM45	exhaustive	weighted	-231.0	$6 \cdot 10^{-5}$
clearcut	log BLOSUM45	hill-climbing	unweighted	-238.3	$1 \cdot 10^{-5}$
quicktree	linear BLOSUM45	exhaustive	unweighted	-269.1	$8 \cdot 10^{-7}$
quicktree	%different	exhaustive	unweighted	-429.1	$1 \cdot 10^{-19}$

Table 3 - CPU time and memory usage for inferring trees for genuine families.

(A) We report the CPU time and memory usage for various methods on each of three alignments. "Distance matrix" reports the time to compute the distances between all N^2 pairs of sequences in the alignment and the space required to store the $N(N-1)/2$ distinct entries of the distance matrix. For BIONJ and clearcut, which do not include the computation of the distance matrix, we added half the time that it takes to compute

the distance matrix. (We use half because the pairs need not be computed both ways.) Values shown with $\approx$ are estimated from the requirements for COG2814 and the scaling behavior of the method (see Methods for details). (B) For each alignment, we show the number of sequences, the number of distinct sequences, the number of columns, and the fraction of positions that are gaps.

(A)

		COG2814		PF00005		16S rRNA	
Program	**Support Values**	**hours**	**GB**	**hours**	**GB**	**hours**	**GB**
FastTree	local	0.05	0.16	0.47	0.3	17.4	2.4
Distance Matrix	–	0.05	0.13	0.71	2.8	49.9	46.5
clearcut	none	0.06	0.22	≈ 1.24	≈ 5.5	≈ 34.7	≈ 54.9
quicktree	none	0.24	0.16	≈ 25	≈ 3.5	$\approx 1,620$	≈ 57.1
quicktree	100 bootstraps	63.5	0.71	$\approx 6,488$	≈ 15.5	$\approx 428,544$	≈ 253.6
BIONJ	none	32.9	0.44	$\approx 4,096$	≈ 10.9	$\approx 129,950$	≈ 109.7
PhyML	local (aLRT)	>330	3.83	–	–	–	–
RAxML	none	>165	0.70	–	–	–	–

(B)

Alignment	**COG2814**	**PF00005**	**16S rRNA**
Type	protein	protein	nucleotide
#Sequences	10,610	52,927	167,547
#Distinct	8,362	39,092	158,022
#Columns	394	214	1,287
%Gaps	10.8%	15.2%	4.3%