SlowTree: a Python Implementation of the FastTree Phylogenetic Tree Construction Algorithm

Dustin Miao Lea Southwick Anthony Xu

April 30, 2025

Abstract

Inferring phylogenetic trees from genetic sequences is fundamental in computational biology, but the classic Neighbor-Joining algorithm [8] can be both slow and memory-intensive on large datasets. To address these limitations, we introduce SlowTree, a pure-Python implementation of the FastTree algorithm [7] that incorporates three key optimizations to dramatically reduce runtime and memory overhead over Neighbor-Joining. In benchmarks on large-scale datasets, SlowTree achieves reasonable accuracy while offering both asymptotic and practical performance gains over Neighbor-Joining. SlowTree is open source and available at `https://github.com/dmiao623/SlowTree`.

1 Introduction

In biology, it is often necessary to construct an evolutionary tree from the data from many life forms. For example, constructing an evolutionary tree from the DNA of several strains of viruses can give us insight into which viruses are related and the possible origins of the viruses. However, to infer an evolutionary tree, we do not know *a priori* the common ancestors of the organisms or the evolutionary pathways, and must infer the phylogeny from only sequence (e.g. DNA or Protein) information. Formally, we would want to solve the **Inferred Phylogeny Problem**, in which we input the N sequences and output a tree with $(2N-1)$ vertices and contain the N species at its leaves with greatest parsomony. A formal mathematical specification of the chosen optimality criterion is presented in Algorithms.

1.1 Neighbor-Joining Algorithm

The algorithm that is commonly used to solve this computational problem is the Neighbor-Joining algorithm, introduced by Saitou and Nei in 1987 [8]. Given sequence information of N species, we initially compute a distance metric as a proxy for phylogenetic proximity. Then, nodes are iteratively joined and distance recomputed, until a single root node remains. The resulting tree is an unrooted binary tree since all leaves have degree one and all internal nodes have degree three (each internal node is connected to its two "child" subtrees and a "parent"). A pseudocode is given in Algorithm 1.

Algorithm 1 Neighbor-Joining Algorithm

procedure NEIGHBOR-JOINING($S_1, S_2, \ldots, S_N$)
    $D \leftarrow N \times N$ pair-wise distance matrix
    $\mathcal{T}^{(0)} \leftarrow$ forest of N isolated leaves
    for $i \leftarrow 1 \ldots N-1$ **do**
        $D^* \leftarrow$ modified D matrix
        $(u, v) \leftarrow$ closest pair of nodes in D^*
        $\mathcal{T}^{(i)} \leftarrow \mathcal{T}^{(i-1)}$ with (u, v) merged into w
        $D \leftarrow$ recomputed D with w.
    end for
    return $\mathcal{T}^{(N)}$
end procedure

The matrix D^* is initialized with the following formula:

$$D^*_{i,j} = \begin{cases} 0 & i = j \\ (n-2)D_{i,j} - \sum_k (D_{i,k} + D_{j,k}) & i \neq j \end{cases} \tag{1}$$

where n is the number of unjoined nodes, and the sum is taken over all nodes k that remain to be joined. When we merge (u, v) into the new node w, the "limb lengths" of u and v are given by the following formulas:

$$D_{u,w} = \frac{D_{u,v} + \delta}{2} \tag{2}$$

$$D_{v,w} = \frac{D_{u,v} - \delta}{2} \tag{3}$$

where

$$\delta = \frac{\sum_k (D_{u,k} - D_{v,k})}{2}. \tag{4}$$

The distances from w to the remaining nodes in the tree are calculated as follows:

$$D_{w,k} = \frac{D_{u,k} + D_{v,k} - D_{u,v}}{2}. \tag{5}$$

In practice, we begin with a multiple-sequence alignment of length L for each of the N species (either DNA or protein sequences). To measure the distance between any two species, we simply count the number of mismatched positions in their pairwise aligned sequences or score their alignment using a chosen similarity matrix.

The time complexity of the Neighbor-Joining algorithm can be analyzed as follows:

- Computing the initial distance matrix takes $\mathcal{O}(N^2 L)$ time, since there are $O(N^2)$ pairs of species, and each one takes $\mathcal{O}(L)$ time to compute the character-wise difference.
- Constructing D^* from D takes $\mathcal{O}(n^2)$ time, where n is the current number of unjoined nodes.

- Finding the best join takes $\mathcal{O}(n^2)$ time, since we just check each pair. Updating distances takes $\mathcal{O}(n)$ time ($\mathcal{O}(1)$ per node).

The total time complexity is thus

$$\mathcal{O}(N^2 L) + \sum_{n=3}^{N} \mathcal{O}(n^2) = \mathcal{O}(N^2(N + L))$$

Although Neighbor-Joining sets a silver standard in terms of accuracy ([6, 1]), the cubic time complexity is insufficient for efficiently computing phylogenies given the current size of genomic databases. For instance, the greengenes database [3] contains the 16S ribosomal rRNA data of 108453 species. Even with increased processing power, Neighbor-Joining fails to adequately scale to datasets of this magnitude.

1.2 A Better Algorithm

The FastTree algorithm, introduced in 2009 by Price, Dehal, and Arkin [7], is an optimized version of Neighbor-Joining that is asymptotically more efficient, uses less memory, and achieves comparable accuracy. We chose to implement the optimizations described in the paper from scratch in Python.

Our implementation of FastTree improves upon Neighbor-Joining in three ways:

1. *Profile-based Computation*: Each root node in the intermediate trees is represented as a profile that aggregates the sequence information of the corresponding leaves. This eliminates the need to compute and maintain a distance matrix in each step.

2. *Top-Hits Heuristic*: Each node maintains a list of $K \approx 2\sqrt{N}$ phylogenetically closest nodes, which reduces the number of merges considered in each step from $\mathcal{O}(N^2)$ to $\mathcal{O}(NK)$.

3. *Integration of Other Algorithms*: We also include heuristical optimizations from previously published works, including the "Weighbor criterion" [2] and the Jukes-Cantor correction [5].

2 Algorithm

2.1 Definitions

A *sequence* is a finite collection of symbols over an *alphabet*. An *alignment* A of N sequences $(s_1, s_2, \ldots, s_N)$, over alphabet Λ is a $N \times L$ matrix with entries in $\Lambda \cup \{\texttt{-}\}$ that minimizes the pairwise-edit distances. Specifically, A_i is equal to s_i interspersed with gap characters (-). Note there exist several efficient algorithms to construct alignments from a set of sequences (for example, [4]).

A *profile* is a space-efficient, probablistic representation of an alignment. Given alignment A defined as above, a profile P is a $|\Lambda| \times L$ matrix where $P_{\lambda,i}$ represents the frequency of character $\lambda \in \Lambda$ in the i-th column of A; gapped positions are ignored. For example, if a

node's leaves consisted of the sequences `ACG`, `AGT`, and `T-T`, then the profile would look as follows:

Character	Position 1	Position 2	Position 3
A	0.67	0	0
C	0	0.5	0
G	0	0.5	0.33
T	0.33	0	0.67

2.2 Problem Statement

Input: An $N \times L$ alignment A of a set of sequences $S = \{s_1, s_2, \ldots, s_N\}$ over an alphabet Λ (e.g. $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$ for DNA) and a function $U : \Lambda' \times \Lambda' \to \mathbb{R}^+$ ($\Lambda' := \Lambda \cup \{\texttt{-}\}$) that quantifies the dissimilarity between two members of the alphabet.

Output: A weighted, rooted tree $\mathcal{T}$ containing $s_1, s_2, \ldots, s_N$ as leaves, where the distance between s_i and s_j approximates the evolutionary distance between species i and j.

2.3 Overview

The algorithm consists of a series of $(N-1)$ sequential steps. In the t-th step, we are given a list of $m = N - t + 1$ tree nodes $T_1, T_2, \ldots, T_m$ and output a list of $m - 1$ tree nodes $T'_1, T'_2, \ldots, T'_{m-1}$, the result of performing one "merge" operation. Initially, each sequence in the alignment is assigned a unique tree node. The result of the algorithm is the tree produced by the $(N-1)$-th step. This is the same overall structure as Neighbor-Joining.

2.4 Profile-based Computation

Each tree node T_u contains a profile P_u. The distance between profiles P_1 and P_2 is computed through

$$D(P_1, P_2) := \sum_{i=1}^{L} \left(\sum_{\lambda_1 \in \Lambda'} \sum_{\lambda_2 \in \Lambda'} (P_1)_{\lambda_1, i} (P_2)_{\lambda_2, i} U_{\lambda_1, \lambda_2} \right), \tag{6}$$

which approximates the pairwise-distance between entries of the corresponding alignments. In matrix form, this is equivalent to

$$D(P_1, P_2) = \operatorname{sum}\left(P_1^T U P_2\right), \tag{7}$$

which is more efficient to compute.

In practice, each tree node is assigned a unique identifier that persists through steps. Distance computations between profiles are cached between steps, ensuring that expensive $\mathcal{O}(L|\Lambda|^2)$ distance computations are done the least number of times.

When merging nodes T_u and T_v into a node T_w, the new profile P_w is constructed as a weighted average of P_u and P_v. Each profile is weighted by the size of the profile (i.e. number of sequences it represents), the proportion of ungapped columns, and a bias from the BIONJ algorithm [2]. This calculation takes $\mathcal{O}(L|\Lambda|)$ and can be vectorized easily.

2.5 Top-Hits Heuristic

Each tree node T_u also maintains a top-hits list $H_u = \{h_{u,1}, h_{u,2}, \ldots, h_{u,m_u}\}$, which approximates the closest nodes to u. The length of a top-hits list m_u varies between steps, but is approximately equal to k, a hyperparameter on the order $\mathcal{O}(\sqrt{N})$. In each step, we only consider merges between nodes in top-hit lists, or approximately $\mathcal{O}(Nk) = \mathcal{O}(N\sqrt{N})$ pairs. We discuss how to initialize and maintain the H_u through each step in subquadratic time.

2.5.1 Top-Hits Initialization

Assuming that the provided distance matrix satisfies metric-space axioms (i.e. $\forall \lambda_1, \lambda_2, \lambda_3 \in \Lambda'$, $U_{\lambda_1,\lambda_2} + U_{\lambda_2,\lambda_3} \geq U_{\lambda_1,\lambda_3}$), the profile distance function defined in Equation 6 satisfies the triangle inequality, implying that for any three profiles P_u, P_v, P_w, the bound of

$$|D(P_u, P_w) - D(P_v, P_w)| \leq D(P_u, P_v) \tag{8}$$

holds. Intuitively, Equation 8 shows that for profiles P_u, P_v sufficiently close to each other, the difference in distance to all other profiles P_w is small, implying that H_u and H_v contains many shared points.

Naively, initialization takes $\mathcal{O}(N^2 L|\Lambda|^2)$, but we can use the observation above to approximate all top-hits lists in subquadratic time.

Algorithm 2 Top-Hits Initialization

```
1: procedure ALL-TOP-HITS(T_1, T_2, ..., T_m)
2:     for u ← 1...m do
3:         if H_u initialized then
4:             Continue
5:         end if
6:         H_u ← TOP-HITS(u)
7:         for v ∈ H_u do
8:             H_v ← H_u
9:         end for
10:    end for
11:    return H_1, H_2, ..., H_m
12: end procedure
```

Algorithm 2 shows one such way. For each uninitialized node, the top-hits are initialized with the brute force algorithm, and all adjacent top-hits lists are updated. This takes on average $\mathcal{O}(N^2 L|\Lambda|^2/K) \approx \mathcal{O}(N\sqrt{N}L|\Lambda|^2)$.

2.5.2 Top-Hits Merging

Suppose nodes T_u and T_v are merged into a node T_w. From Equation 8, we can upper-bound the difference between $D(P_u, P_w)$ and $D(P_v, P_w)$, suggesting that H_w is similar to H_u and H_v. Thus, H_w can be easily initialized as $H_u \cup H_v$. Meanwhile, for all nodes $x \in \{1, \ldots, m\} \backslash \{u, v\}$

such that $u \in H_x$ or $v \in H_x$, we replace u or v with w, since u and v have been joined into w.

Each merge operation potentially reduces the size of H_x by 1, so to maintain a sufficiently large buffer of potential merges, we reinitialize the top-hits using Algorithm 2 every $R \sim \mathcal{O}(\sqrt{n})$ steps.

In practice, Top-Hits lists are implemented using hash-tables (Python's `set`) that support efficient iteration, retrieval, and merging. The reinitialization step is done lazily by tracking current ancestors with a union-find data structure, which offers a significant constast-time speed up.

2.6 Integration of Other Algorithms

2.6.1 Weighbor Criterion

In the BIONJ algorithm [2], each distance $D(i,j)$ is accompanied by an estimated variance $\sigma^2_{i,j}$, and merges are performed by inverse–variance weighting of distance estimates. The FastTree adopts a modified version of this framework, adapted for profile-based computation.

For any two profiles P_u and P_k, we computes the mean and second moment of the dissimilarity at column i:

$$\mu_i = \sum_{\lambda_1,\lambda_2 \in \Lambda} (P_u)_{\lambda_1,i}\,(P_k)_{\lambda_2,i}\,U_{\lambda_1,\lambda_2}, \quad m_i = \sum_{\lambda_1,\lambda_2 \in \Lambda} (P_u)_{\lambda_1,i}\,(P_k)_{\lambda_2,i}\,U^2_{\lambda_1,\lambda_2}. \tag{9}$$

The variance of the profile–profile distance is then

$$\sigma^2_{u,k} = \sum_{i=1}^{L} \left(m_i - \mu_i^2\right), \tag{10}$$

which captures the uncertainty in $d(u,k) = \sum_i \mu_i$ arising from the profile counts. Then, if nodes T_u and T_v are to be merged into T_w, for each remaining node T_k, the inverse–variance weights are defined as

$$\alpha_u = \frac{1/\sigma^2_{u,k}}{1/\sigma^2_{u,k} + 1/\sigma^2_{v,k}}, \quad \alpha_v \;=\; 1 - \alpha_u. \tag{11}$$

FastTree then sets

$$d(w,k) = \alpha_u\, d(u,k) \;+\; \alpha_v\, d(v,k), \tag{12}$$

$$\sigma^2_{w,k} = \left(\tfrac{1}{\sigma^2_{u,k}} + \tfrac{1}{\sigma^2_{v,k}}\right)^{-1}. \tag{13}$$

Because α_u is larger when $\sigma^2_{u,k}$ is smaller, the new distance $d(w,k)$ is dominated by the more reliable (lower-variance) branch length. This schema criterion topological errors by down-weighting noisy, high-variance distance estimates in each merge.

2.6.2 Jukes-Cantor Correction

The Jukes–Cantor (JC) model introduces a log-linear correction to convert an observed proportion of nucleotide differences into an estimated number of substitutions, thereby compensating for multiple unseen substitutions at the same site. In practice, FastTree computes the final branch length between two sequences or profiles using the JC formula:

$$d_{\text{corrected}} := -\frac{3}{4} \ln \left(1 - \frac{4}{3} d_{\text{raw}} \right), \tag{14}$$

where d_{raw} is the raw distance based on the average dissimilarity between two profiles and proportion of gapped columns. For peptide sequences, a similar correction from the FastTree paper based on empirical results was used:

$$d_{\text{corrected}} := -1.3 \ln \left(1 - d_{\text{raw}} \right). \tag{15}$$

In both cases, this transformation serves to correct the observed fraction of differences into a more accurate estimate of the true evolutionary distance and thus avoiding the systematic underestimation of branch lengths when sequences are highly diverged.

3 Results

3.1 Setup

We ran our code on a multiple alignment of 16S ribosomal rRNA from the greengenes database [3]. (Specifically, the multiple alignment used was `core_set_aligned.fasta`, which has 4938 sequences of length 7682.) The algorithms that were benchmarked were:

- The Neighbor-Joining algorithm, implemented in Python from scratch.
- An algorithm where we simply joined nodes randomly, which was implemented as an accuracy benchmark. We called this algorithm "Random-Joining."
- Our version of FastTree, implemented in Python.
- The official version of FastTree, from FastTree's website.

Each of the algorithms were programmed to take as input a multiple alignment of nucleotide sequences, and return a tree in Newick format.

The algorithms were run on the cluster computers on the fifth floor of the Gates-Hillman Center, and their runtime and peak memory usage were measured. To compare results for different sizes of datasets, a script was created to sample $n < N$ sequences and $c < L$ columns from a multiple alignment of N sequences, each of length L.

The accuracies of the algorithms were measured using `CompareTree.pl` from the official FastTree GitHub repository, which is a Perl script that counts the number of splits that are shared between two trees. (A split is the two sets of leaves on either side of an edge.)

3.2 Runtime

For the testing, we fixed $c = 2000$, and we tested the values $n = 100$, 500, 1000, 1500, and 2000. For each of these values, we used the script mentioned earlier to sample n sequences and c columns from `core_set_aligned.fasta`, then ran SlowTree and Neighbor-Joining on each of the sampled alignments. (We ran the official version of FastTree as well, but the difference in runtime was too extreme to be included in the results. Despite having the same time complexity as SlowTree, the official FastTree ran about 100 times faster because of its highly optimized C source code.)

We can see from Figure 1 that SlowTree runs faster than Neighbor-Joining when n is large, and that the difference becomes larger as n increases. This makes sense, since for a multiple alignment with n sequences and c columns, the Neighbor-Joining algorithm has a time complexity of $\mathcal{O}(n^2(n+c))$, while SlowTree's time complexity is $\mathcal{O}(n\sqrt{n}c|\Lambda|^2)$. Thus, when $c \approx n$, SlowTree is a factor of $\mathcal{O}(\sqrt{n}/|\Lambda|^2)$ faster than Neighbor-Joining. For larger datasets, we would have $c \ll n$, and the difference between SlowTree and Neighbor-Joining would grow even larger.

Figure 1: Runtime of SlowTree and Neighbor-Joining on length-2000 sequences.

In theory, the distance between SlowTree and Neighbor-Joining should be even more pronounced than what is shown here. However, in practice, SlowTree's constant factor is quite significant, mainly because of the creation of many Python objects, which slow down the code because of the large number of heap allocations. Meanwhile, Neighbor-Joining is simply running arithmetic computations. Therefore, SlowTree only starts to overtake Neighbor-Joining around $n = 1500$. Most datasets used in computational biology have far more than 1500 sequences, so in those cases, SlowTree is still faster.

3.3 Memory

We can see a similar trend in the memory usage in Figure 2: SlowTree's memory usage is less than that of Neighbor-Joining for large n, with the difference growing larger as n increases.

This also makes sense, because the Neighbor-Joining algorithm requires the entire distance matrix to be stored at each step, so the memory usage is $\mathcal{O}(n^2)$. In contrast, SlowTree only needs to store the profiles and top-hits lists of every node, so the memory usage is $\mathcal{O}(nc|\Lambda| + n\sqrt{n})$, which is less than $\mathcal{O}(n^2)$ for large n.

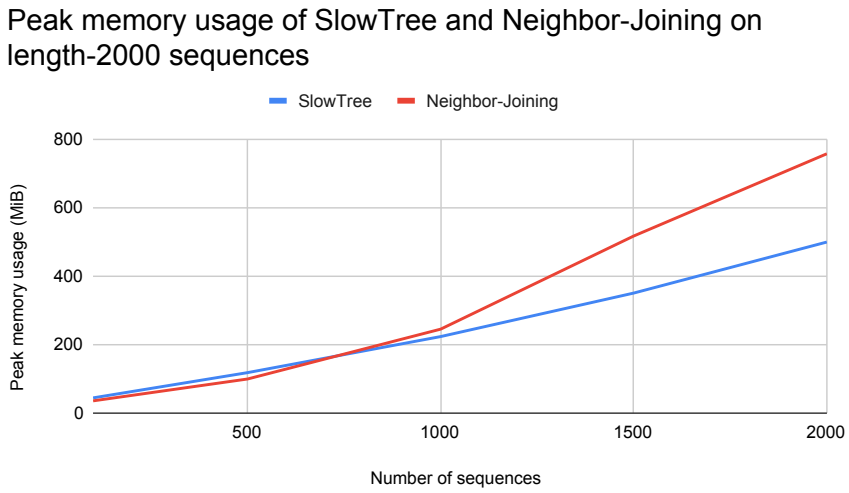

Figure 2: Peak memory usage of SlowTree and Neighbor-Joining on length-2000 sequences.

Notably, when $n \approx c$, one might expect that Neighbor-Joining and SlowTree use comparable amounts of memory, but in practice, SlowTree uses less. This is likely because Neighbor-Joining still needs to calculate the distances between all sequences at the beginning of the algorithm, which requires loading all sequences into memory just as SlowTree does.

3.4 Accuracy

We present confusion matrices for the accuracies for $n = 500$ and $n = 2000$ in Figure 3, where the accuracy is given by the fraction of splits that are shared between two trees.

Note that while SlowTree is more efficient for higher n, the accuracy remains constant. We can also see that SlowTree significantly outperformed the Random-Joining algorithm in accuracy (when the accuracy is measured against either Neighbor-Joining or the official FastTree), indicating that SlowTree's accuracy is not due to random chance.

4 Future Work

While we have implemented the core of the FastTree algorithm here, there are several optimizations referenced in the original paper to further optimize time and accuracy. We discuss two such optimizations.

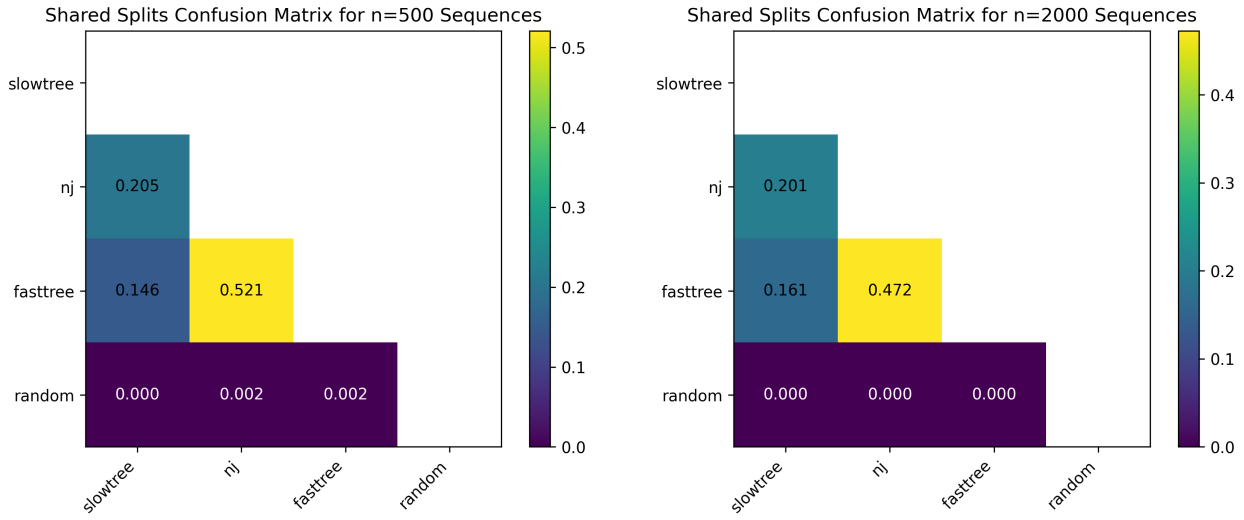

Figure 3: Confusion matrices of SlowTree, Neighbor-Joining, FastTree, and Random Joining for an alignment of 500 sequences (left) and 2000 sequences (right).

4.1 Nearest Neighbor Interchanges

The current implementation of SlowTree directly returns the tree obtained after completing the $(N-1)$ merge operations. However, the accuracy of this tree can be refined with Nearest Neighbor Interchanges (NNIs). Each NNI exchanges two adjacent subtrees to marginally increase the structure of the tree. In each round of NNIs, FastTree started with children nodes and worked inwards. At each step, FastTree either kept the original topology $((A,B),(C,D))$, or changed the local structure of the tree to $((A,C),(B,D))$ or $((A,D),(B,C))$, depending on which topology minimized the total distance between the two pairs of adjacent nodes.

4.2 Caching Nearest Neighbor Computations

Instead of iterating through all $\mathcal{O}(N\sqrt{N})$ comparisons in each step, we can maintain unchanged comparisons made in previous rounds or perform several non-intersecting merges in parallel, which reduces the number of expensive distance computations.

References

[1] Kevin Atteson. *The performance of neighbor-joining algorithms of phylogeny reconstruction*, volume 1276, pages 101–110. 11 2006.

[2] William J. Bruno, Nicolas D. Socci, and Alan L. Halpern. Weighted neighbor joining: A likelihood-based approach to distance-based phylogeny reconstruction. *Molecular Biology and Evolution*, 17(1):189–197, 2000.

[3] Todd Z DeSantis, Philip Hugenholtz, Neils Larsen, Mark Rojas, Eoin L Brodie, Keith Keller, Thomas Huber, Daniel Dalevi, Ping Hu, and Gary L Andersen. Greengenes, a Chimera-checked 16S rRNA gene database and workbench compatible with ARB. *Applied and Environmental Microbiology*, 72(7), 04 2006.

[4] R.C. Edgar. MUSCLE: multiple sequence alignment with improved accuracy and speed. In *Proceedings. 2004 IEEE Computational Systems Bioinformatics Conference, 2004. CSB 2004.*, pages 728–729, 2004.

[5] H. Jukes, T. and R. Cantor, C.˙Evolution of protein molecules. In N. Munro, H. editor, *Mammalian Protein Metabolism*, pages 21–132. Academic Press, New York, 1969.

[6] Radu Mihaescu, Dan Levy, and Lior Pachter. Why neighbor-joining works. *Algorithmica*, 54(1):1–24, May 2009.

[7] Morgan N. Price, Paramvir S. Dehal, and Adam P. Arkin. FastTree: Computing large minimum evolution trees with profiles instead of a distance matrix. *Molecular Biology and Evolution*, 26(7):1641–1650, 2009.

[8] Naruya Saitou and Masatoshi Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406–425, 07 1987.