Identifying the Differences Between FNJ and DNJ

Simon Ekman

Abstract

Two algorithms for reconstructing phylogenetic trees, Fast Neighbor Joining (FNJ) and Dynamic Neighbor Joining (DNJ), are examined in this thesis. The two algorithms build on the same concept, posing the question of whether they are distinct algorithms. To closer investigate this, the differences and similarities between the algorithms are highlighted, and they are discussed theoretically as well as tested practically. Although a definitive answer to the question is not given here, it will be shown that the two algorithms achieve the same results and build on the same concept, while achieving the same practical runtime.

1. Introduction

In the field of phylogenetics, a common problem is identifying the evolutionary history of a set of genomic sequences and how they are related to each other. For this purpose, it is desirable to construct a phylogenetic tree, where sequences of currently living species are represented as leaves and those of ancestral species as other vertices. Most often the only information available are the genomes of currently living species, while those of ancestral species are unknown. On a conceptual level, the problem is to accurately reconstruct a tree with weighted edges when the only information available are the distances between its leaves.

Several algorithms have been proposed to solve this problem. One such algorithm is the Neighbor Joining algorithm (NJ), which has given rise to many derivative algorithms. This thesis examines two such algorithms, namely Fast Neighbor Joining (FNJ) and Dynamic Neighbor Joining (DNJ). FNJ was proposed by I. Elias and J. Lagergren in 2009, while DNJ was proposed by P.T.L.C. Clausen in 2023. Both algorithms will be described in detail in the definitions section. Moreover, the premise of DNJ is similar enough to FNJ that it poses the question of whether they are distinct algorithms. The main objective of this thesis is to investigate this closer. In the methods section, the two algorithms will be discussed theoretically, as well as tested practically.

2. Definitions

2.1. Preliminaries

A weighted tree $T(V, E)$ is an acyclic graph with a mapping $W_T : E \to \mathbb{R}$ that assigns every edge in E a weight. A vertex $v \in V$ in a tree is called a leaf if it is only adjacent to one other vertex. Two leaves u, v are called neighbors (also called siblings) if they share an adjacent vertex. A weighted tree T is a *phylogenetic tree* if all edges are strictly positive-weighted.

A *distance matrix* D is a symmetric matrix with 0-valued diagonal elements. If there exists a phylogenetic tree $T(V, E)$ such that $D_{ij} = \sum_{e \in P} W_T(e)$ for the unique path P between any two leaves i, j, then D is called *additive* and is said to be realized by T.

A tree $T^*(V, E)$ is called *starlike* if there are $|V| - 1$ leaves and a single root connecting all leaves. Although phylogenetic trees are unrooted, NJ algorithms consider an implicit root r of an initially starlike tree until the reconstruction is complete. In the context of NJ algorithms, vertices are referred to as taxa.

2.2. Neighbor Joining

The goal of Neighbor Joining [5] and its derivative algorithms is as follows: starting with a starlike tree T^*, reconstruct a tree T using its distance function D_T. This is done by iteratively identifying and joining pairs of taxa that are the most likely to be neighbors. Every iteration can be divided into two parts:

an *optimal join search* and an *updating step.*

NJ algorithms use a measure called the NJ sum (also called NJ function [5, 3] or join criterion [2]):

$$S_{ij} := (n-2)D_{ij} - R_i - R_j$$

for a pair of taxa i, j, where n is the number of taxa in the current iteration and R_i is the sum of the ith row in the distance matrix D [6, 3, 2]. Intuitively, it is a measure of the loss from joining i, j as neighbors. In the optimal join search, NJ algorithms use a greedy approach where the pair x, y with the least loss are considered the most likely neighbors. After this pair is identified, the two taxa are joined as neighbors by creating a new vertex z that connects to x, y and the implicit root r. Consequently, the vertices x, y are discontinued as taxa as z replaces them.

In the updating step, D is updated with inferred distances from z to every other taxon:

$$D'_{ij} = \begin{cases} D_{ij} & z \notin \{i, j\} \\ \frac{D_{xj}+D_{yj}-D_{xy}}{2} & z = i \neq j \\ \frac{D_{xi}+D_{yi}-D_{xy}}{2} & z = j \neq i \end{cases},$$

where D' is the matrix that will be used in the following iteration, and what D becomes after being updated [6, 2].

This process is repeated until 3 taxa remain, at which point the reconstruction is complete. The edge weights for the remaining three edges incident to r can be computed by solving a system of equations:

$$\begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} W_T(r,a) \\ W_T(r,b) \\ W_T(r,c) \end{bmatrix} = \begin{bmatrix} D_{ab} \\ D_{ac} \\ D_{bc} \end{bmatrix},$$

where a, b, c are the remaining three taxa.

To reduce runtime, R can be initialized as a set of row sums at the start, and is updated alongside D:

$$R'_i = \sum_j D'_{ij} = R_i - D_{xi} - D_{yi} + D_{zi},$$

after joining x, y as z [6]. This allows for NJ sums to be computed in constant time.

Neighbor Joining runs in $\mathcal{O}(N^3)$ time, where N is the initial number of taxa. It performs $N-3$ iterations to reduce the number of taxa to 3, and within those it iterates over an $\mathcal{O}(n^2)$ number of taxon pairs during the optimal join search, with n being the number of taxa in a given iteration. Updating D and R in the updating step is done in $\mathcal{O}(n)$ time, while computing S_{ij} is done in $\mathcal{O}(1)$ time.

2.3. Fast Neighbor Joining

Fast Neighbor Joining [3] introduces the concept of visibility, which it uses to reduce the number of pairs it needs to iterate over.

A pair i, j is *visible* from i with respect to D if $j = \text{argmin}_j S_{ij}$. A pair i, j is visible with respect to D if it is visible from either i or j. We will use $\mathcal{V}_D(i)$ to denote a pair visible from i with respect to D, and $S_{\mathcal{V}_D(i)}$ to denote the NJ sum of said pair.

At the start, FNJ initializes a visible set $\mathcal{V}$ that contains a pair $\mathcal{V}_D(i)$ for every taxa i. In the optimal join search, it then iterates over $\mathcal{V}$ instead of every pair. Like with D, $\mathcal{V}$ also needs to be updated in the updating step. This is done after updating D, R:

$$\mathcal{V}' = (\mathcal{V} \backslash (i, j) : \{i, j\} \cap \{x, y\} \neq \emptyset) \cup \{\mathcal{V}_{D'}(z)\}.$$

When a pair x, y is joined, FNJ drops any other pairs involving x or y from $\mathcal{V}$, and in some cases this will lead to the visible pairs for some taxa not being present. FNJ should always detect an optimal join regardless: for a pair i, x dropped from $\mathcal{V}$ after joining x, y as z, in the following iteration we will have that $\mathcal{V}_D(i) = (i, z)$ given that D is additive. The new taxon z would otherwise appear to be further away from i than x, meaning that the edge between x and z would not be strictly positive-weighted. In fact, the same reasoning may be applied for certain cases when D is not additive:

Given a distance matrix D, if there exists an additive distance matrix D_T realized by a phylogenetic tree T such that $|D - D_T|_\infty < \frac{\mu}{2}$, where μ is the minimum edge weight of T, then D is called *nearly additive* with respect to D_T [1, 3]. For such a distance matrix, if i, z are neighbors in the tree realized by the additive D_T, then $\mathcal{V}_D(i) = (i, z)$ according to Lemma 2 in Elias, Lagergren [3].

The rest of the procedure is identical to NJ.

The visible set $\mathcal{V}$ contains an $\mathcal{O}(n)$ number of elements in every iteration, allowing an optimal join search in $\mathcal{O}(n)$ time, and can be updated in $\mathcal{O}(n)$ time as well. This gives FNJ an $\mathcal{O}(N^2)$ runtime overall.

2.4. Dynamic Neighbor Joining

Before describing Dynamic Neighbor Joining, we define pseudo-visibility:

A pair i, j is *pseudo-visible* from i with respect to D if $j = \text{argmin}_{j<i} S_{ij}$. The pseudo-visible pair of i will be denoted $\widetilde{\mathcal{V}}_D(i)$.

Dynamic Neighbor Joining [2] bases itself on the following lemma:

Lemma 1, Clausen. Given a distance matrix D, row sums R, NJ sums S, and their updated variants D', R', and S' used in the following iteration:

$$S_{\mathcal{V}_D(i)} = (n-2)D_{ij} - R_i - R_j \leq (n-3)D'_{ik} - R'_i - R'_k = S'_{ik},$$

for $\mathcal{V}_D(i) = (i, j)$, where $k \notin \{i, z\}$. In other words, the NJ sum $S_{\mathcal{V}_D(i)}$ will be less than or equal to the NJ sum of any pair involving the taxon i in the following iteration, provided the newly formed taxon z does not make a better join when paired with i [2].

$\square$

Dynamic Neighbor Joining uses a set Q of estimated minimal NJ sums per taxon. At the start, Q is initialized with $Q_i = S_{\mathcal{V}_D(i)}$ for every taxon i. The optimal join search is more involved and proceeds as follows:

For every taxon i, check whether $M > Q_i$, where M is the NJ sum of the best join encountered since the beginning of the search. If it is, search all pairs i, j where $j < i$ and assign $Q_i = S_{\widetilde{\mathcal{V}}_D(i)}$; otherwise proceed to the next taxon without searching through pairs. Note that only pairs i, j for which j is lower-indexed than i are searched: this is because DNJ searches through Q back-to-front, so any pair i, j for which $j > i$ will already have been covered since examining j.

In short, the algorithm only iterates over pairs i, j in the optimal join search where $j < i$ and $\widetilde{\mathcal{V}}_D(i)$ has a possibility, as indicated by Q_i, of being a better join than the so far encountered best join.

After joining x, y as z and updating D, R at the end of an iteration, Q is updated as well:

$$Q'_i = \begin{cases} \min\{S_{iz}, Q_i\} & i > z \\ \min_{i<z} S_{iz} & i = z \\ Q_i & i < z \end{cases} .$$

However, if z is assigned the last position in the list of taxa (as was done in this implementation), this update can be omitted for all $i \neq z$ as all other taxa will be lower-indexed. The intuition for only updating Q_i for which $i > z$ is the same as in the optimal join search, namely that Q_z will cover all pairs i, z where $i < z$.

Unlike D, R and $\mathcal{V}$, Q is for the most part only updated in the optimal join search. If an update is applied to Q for $i \neq z$ in the updating step, it is only done to satisfy the conditions of lemma 1, and does not check any pairs other than i, z. The lemma is relevant for this purpose, as any Q_i for which $M \leq Q_i$ will not be examined closer and thus not updated. Regardless, DNJ performs the optimal join search correctly: the lemma shows that a Q_i that was not updated in the previous iteration will not be greater than any NJ sum involving that taxon in the current iteration, so if $M \leq Q_i$, then $M \leq S_{\mathcal{V}_D(i)}$.

Aside from the use of Q, the rest of the procedure is identical to NJ.

The set Q allows an optimal join search in $\mathcal{O}(dn)$ time, where d is the number of taxa that are examined closer during an optimal join search. The set Q in the implementation used here is updated in $\mathcal{O}(1)$ time at the end of an iteration, giving the algorithm an $\mathcal{O}(dN^2)$ runtime overall. It holds that $1 \leq d < n$ for all d, but their relation outside of that depends on how the taxa are ordered, making it difficult to determine the runtime purely in terms of N.

3. Methods

3.1. Discussion

Readers may have noticed that the concept of visible pairs is used in both FNJ and DNJ. While FNJ initializes a set of visible pairs $\mathcal{V}$ for all taxa, DNJ initializes a set of NJ sums given by the same visible pairs. Conversely, FNJ finds the NJ sum of elements in $\mathcal{V}$ while DNJ finds visible pairs representing elements in Q during the optimal join search. Effectively, both algorithms identify the optimal join by finding visible pairs and NJ sums, albeit in opposite order.

As shown by Atteson [1, 3], the phylogenetic tree that induces an additive or nearly additive distance matrix is unique, and NJ is able to correctly determine the topology of the unique phylogenetic tree. Given that FNJ and DNJ are both able to successfully identify the optimal join given an additive or nearly additive distance matrix, these algorithms will also determine the correct, and more importantly, same topology.

The time complexity of FNJ and DNJ may be the greatest difference. FNJ has a consistent $\mathcal{O}(N^2)$ runtime, while DNJ runs in $\mathcal{O}(dN^2)$. The unpredictable nature of d makes the two algorithms difficult to compare theoretically in terms of time complexity, and as such further discussion is left to the results section.

3.2. Simulation

FNJ and DNJ were implemented and tested in Python 3.11. Two tests were run to more closely determine how the runtimes of the algorithms were related. Specifically, d needed to be practically examined to determine how it related to N.

Runtimes were measured by performing the algorithms on randomly generated additive distance matrices with increasing numbers of taxa. To ensure additive distance matrices, random phylogenetic trees were constructed starting with starlike trees with given numbers of taxa. Taxa were joined as neighbors at random, and edges were assigned random weights ranging from 2 to 12. The distance matrices were then derived from the generated phylogenetic trees.

For every number of taxa, the same distance matrix was used for both FNJ and DNJ. The time it took to generate the distance matrix was not measured, only the time it took for FNJ and DNJ to reconstruct the tree.

The first test was run with $N = \left\lfloor 250 \cdot 1.16^{(k-1)} \right\rfloor$ taxa in step k, for $1 \leq k \leq 27$, while the second test had $N = 500k$ taxa in step k, for $1 \leq k \leq 24$.

4. Results

Runtime results are shown directly in fig. 1, and also per N^2 and $N^2\sqrt{N}$, for N taxa in fig. 2 and fig. 3.

Early runtime tests on FNJ and DNJ with smaller numbers of taxa showed results that pointed towards d scaling logarithmically. The first test was con-

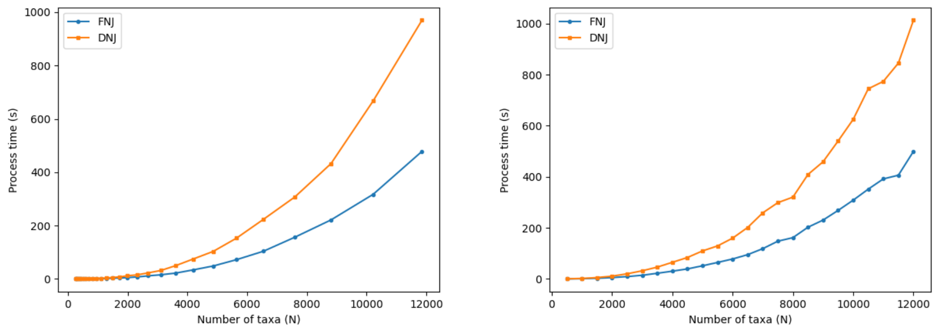

Fig. 1: result of the first test (left) and the second test (right).

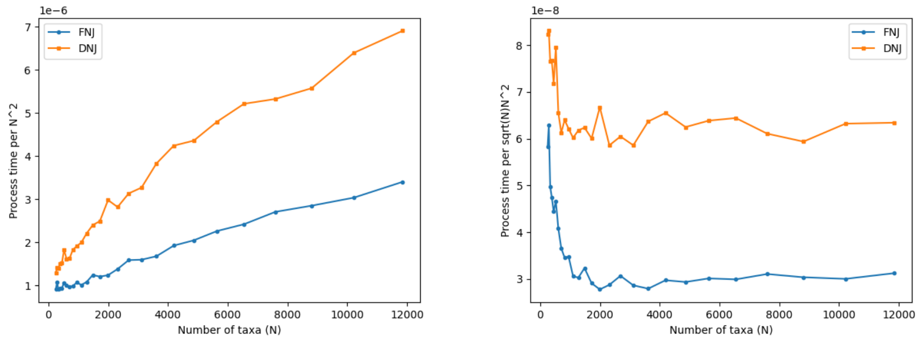

Fig. 2: result of the first test per N^2 taxa (left) and $N^2\sqrt{N}$ taxa (right).

figured to closer investigate this hypothesis. Ultimately this initial hypothesis was discarded, as the results as seen in fig. 2 and fig. 3 instead suggest that the runtime per $N^2\sqrt{N}$ taxa varies within a fixed interval for large N. This suggests that both implementations run in $\mathcal{O}(N^2\sqrt{N})$, and that d would be an $\mathcal{O}(\sqrt{N})$ variable in practice.

These results conflict with the ones presented by Clausen [2]. In Clausen's study, FNJ is shown to be consistently slower than DNJ, whereas the opposite is shown here, and furthermore have a runtime almost identical to NJ, which would suggest an $\mathcal{O}(N^3)$ time complexity. In addition, Clausen states in his results that d achieves an $\mathcal{O}(1)$ scaling.

Clausen implemented DNJ in C and compared it with FNJ through Fast-phylo, where FNJ is implemented in C++ , as well as a C implementation of NJ, among other NJ-based algorithms [4, 2]. In this study, both FNJ and DNJ were implemented in Python. The inconsistency between programming languages in Clausen's study may be a reason why FNJ is shown to underperform there, while it comes closer to expected results in this study.

Additionally, Clausen's implementation of DNJ is designed to be more op-

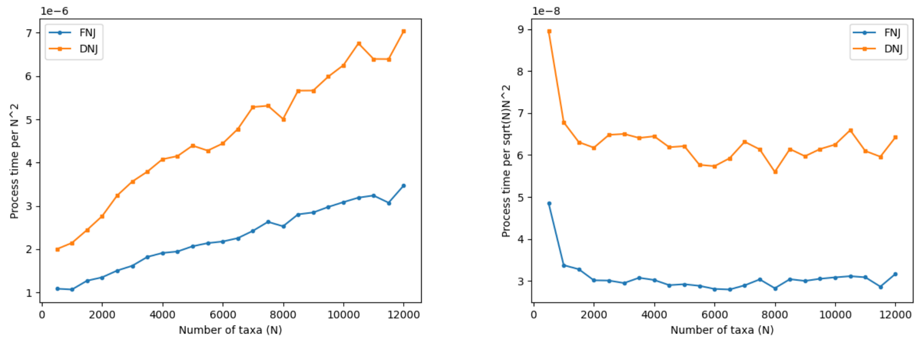

Fig. 3: result of the second test per N^2 taxa (left) and $N^2\sqrt{N}$ taxa (right).

timized in terms of memory usage. This may have had an impact on runtime also.

Lastly, there is a possibility that d is actually $\mathcal{O}(1)$ and that DNJ's $\mathcal{O}(N^2\sqrt{N})$ runtime is caused entirely by other factors, as test results on FNJ unexpectedly show an $\mathcal{O}(N^2\sqrt{N})$ runtime as well.

In reality, there is no formal definition for equivalence between algorithms. Although it has been shown that FNJ and DNJ build on the same concept and achieve the same runtime, a clear answer to whether they are the same algorithm cannot be given in this thesis.

References

[1] Kevin Atteson. The performance of neighbor-joining methods of phylogenetic reconstruction. *Algorithmica*, 25:251–278, 1999.

[2] Philip TLC Clausen. Scaling neighbor joining to one million taxa with dynamic and heuristic neighbor joining. *Bioinformatics*, 39(1):btac774, 2023.

[3] Isaac Elias and Jens Lagergren. Fast neighbor joining. *Theoretical Computer Science*, 410(21-23):1993–2000, 2009.

[4] Mehmood Alam Khan, Isaac Elias, Erik Sjölund, Kristina Nylander, Roman Valls Guimera, Richard Schobesberger, Peter Schmitzberger, Jens Lagergren, and Lars Arvestad. Fastphylo: Fast tools for phylogenetics. *BMC Bioinformatics*, 14(1):1–9, 2013.

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

[6] James A Studier and Karl J Keppler. A note on the neighbor-joining algorithm of saitou and nei. *Molecular Biology and Evolution*, 5(6):729–731, 1988.