Neighbour Joining Method (Saitou and Nei, 1987)

Summary

The Neighbour Joining method is a method for re-constructing phylogenetic trees, and computing the lengths of the branches of this tree. In each stage, the two nearest nodes of the tree (the term "nearest nodes" will be defined in the following paragraphs) are chosen and defined as neighbours in our tree. This is done recursively until all of the nodes are paired together.
The algorithm was originally written by Saitou and Nei, 1987. In 1988 a correction for the paper was sent by Studier & Keppler. The correction was first of all for the proof of the algorithm, and second of all made a slight change to the algorithm which brought the efficiency down to $O(n^3)$ (as is explained in the following paragraphs). We will first of all describe the original algorithm, and then elaborate on the changes made by Studier & Kepler.

What are neighbours?
Neighbours are defined as a pair of OTU's (OTU=operational taxonomic units, or in other words – leaves of the tree), who have one node connecting them.
For instance, in the tree in figure 1, nodes A and B are neighbours (connected by only one internal node), and nodes C and D are neighbours, whereas nodes A and C (for example) are not neighbours.

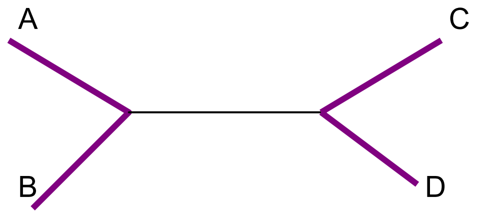


Figure 1

How do we find neighbours, and how de we construct our tree?

1. We start off with a star tree:

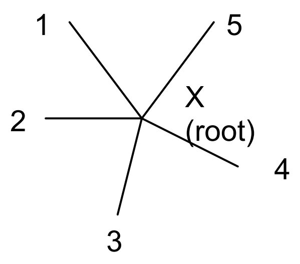


Figure 2(a)

2. We define some kind of distance parameter between our nodes (1 through 5), and enter this parameter into a distance matrix. (see following paragraphs). The columns and rows of the matrix represent nodes, and the value i,j of the matrix represent the distance between node i and node j. Note that the matrix is symmetric, and that the diagonal is irrelevant, therefore only the top half (or lower half) are enough.
3. We pick the two nodes with the lowest value in the matrix defined in step 2. These are defined as neighbours.

For example, assuming nodes 1 and 2 are the nearest, we define them as neighbours.

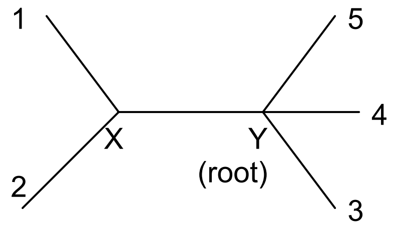


Figure 2(b)

4. The new node we have added is defined as node X.
5. We now define the distance between node X and the rest of the nodes, and enter these distances into our distance matrix. We remove nodes 1 and 2 from our distance matrix.
6. We compute the branch lengths for the branches that have been joined (for figure 2(b), these are branches 1-X and 2-X) .
7. We repeat the process from stage 2 – once again we look for the 2 nearest nodes, and so on.

… And now in detail

Additive Trees

The algorithm and the proofs here are all based on the assumption that we are dealing with additive trees. An additive tree means a tree where, for instance (figure 2(b)):

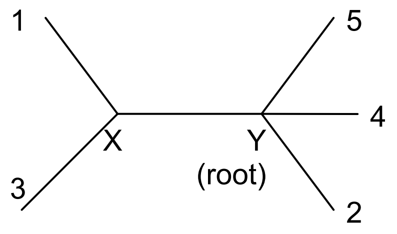
the distance between nodes 1 and 2	=	the distance between node 1 and node X	+	the distance between node 2 and node X

What does the term "nearest nodes" mean?

In the original article, Saitou and Nei defined the two nearest nodes as the pair of nodes that give the *minimal sum of branches* when placed in a tree.

First of all – some notations:
D_{ij} is defined as the distance between leaves i and j (this is the distance which we have as an input from our distance matrix).
L_{xy} is defined as the sum of branch lengths between node X and node Y. L is used as a notation for distances between internal nodes, or an internal node to a leaf. For instance, in figure 2(b) L_{1X} is the distance between the internal node X and leaf (node) 1.

So what we need to compare in each round, (stage 3 above) is the sum of branches for each pair of nodes. That means we compare the sum of branches when 1 and 2 are neighbours (figure 2(a)), which we notate as S_{12}, to the sum of branches when 1 and 3 are neighbours (figure 3), which we notate as S_{13}, etc., etc. So we're looking for the minimal S_{ij} where i and j are numbers of nodes, and $i<j$.



How do we compute this S_{ij}?
For instance:

$$S_{12} = L_{XY} + (L_{1X} + L_{2X}) + \sum_{i=3}^{N} L_{iY} \qquad (1)$$

The problem is, that the terms in this equation are "L terms" (L_{XY}, L_{1X}, etc.), but our input in the algorithm is always a "D term" (D_{12}, D_{23}, etc.). Therefore we have to find a way to convert the "L terms" into D_{ij} terms. This is exactly what is done in the original paper, and is summarized here:

Let's deal with the first term in equation (1) – L_{XY}:

L_{XY} (for figure 2(b)) is given by:

$$L_{XY} = \frac{1}{2(N-2)}[\sum_{k=3}^{N}(D_{1k}+D_{2k})-(N-2)(L_{1x}+L_{2x})-2\sum_{i=3}^{N}L_{iY}] \qquad (2)$$

The first term in this equation is the sum of all distances which include L_{XY}, and the other terms are for excluding irrelevant branch lengths
(the best way to understand this equation is to draw the tree in figure 2(b); then mark a (+) next to each branch which is added in the equation, and a (-) next to each branch that is removed, i.e. minused, in the equation. Add up all the (+)'s and (-)'s... at the end you'll have exactly L_{XY} left).

So for L_{XY} we've partially replaced "L terms" with "D terms". We'll see that the remaining "L terms" in this equation are cancelled out by "L terms" from the other terms in equation 1.

The second and third terms of equation 1 are disposed of simply:
$L_{1X} + L_{2X} = D_{12}$ (3)
(Reminder – this is because our tree is additive)

And finally, the fourth term of equation 1 is replaced as follows:

$$\sum_{i=3}^{N}L_{iY} = \frac{1}{N-3}\sum_{3\le i<j}D_{ij} \qquad (4)$$

So if we replace all these terms in equation 1 we get:

$$S_{12} = L_{XY} + (L_{1X} + L_{2X}) + \sum_{i=3}^{N}L_{iY} = \frac{1}{2(N-2)}\sum_{k=3}^{N}(D_{1k}+D_{2k}) + \frac{1}{2}D_{12} + \frac{1}{N-2}\sum_{3\le i<j}D_{ij}$$

(5)

So this gives us all the information we need for stage 2 of the first round of the algorithm.

Calculating the new distances

What we still have to do is to compute the distance between the new node we created and the rest of the leaves (stage 5). Let's name the new node X created in figure 2(b) as node (1-2). So we get:

$$D_{(1-2)j} = (D_{1j}+D_{2j})/2 \qquad (3\le j\le N) \qquad (6)$$

So now our matrix (stage 2) is enlarged by 1, and then after deleting the data for nodes 1 and 2, will be reduced by 2 – altogether, the matrix is reduced by one.

Computing the branch lengths

Once again, using the example in figure 2(b), we estimate L_{1X} and L_{2X} as follows:

$L_{1X} = (D_{12}+D_{1Z}-D_{2Z})/2$
$L_{2X} = (D_{12}+D_{2Z}-D_{1Z})/2$

Where $D_{1Z} = (\sum_{i=3}^{N} D_{1i})/(N-2)$ *and* $D_{2Z} = (\sum_{i=3}^{N} D_{2i})/(N-2)$

Z represents the group of all leaves except leaves 1 and 2. This is as if Z represents the *whole* cluster of these leaves, as intuitively drawn in figure 4

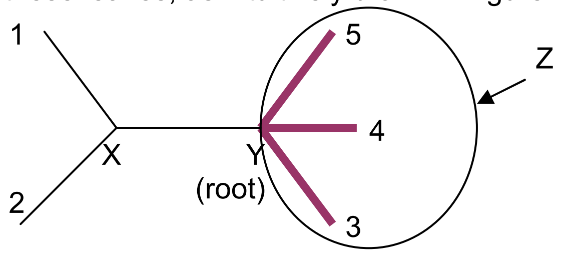


figure 4

What changed between the original algorithm and Studier & Keppler:

In the original algorithm, the parameter used to chose which two nodes are neighbours is the *minimal sum of branches*, Sij.
Studier and Keppler offered an alternative parameter. This parameter, designated as Mij, is actually a transformation of Sij. Mij gives us complexity of O(n^3), whereas Sij can only give us an O(n^5) complexity. As well, it is proven that S and M are related criteria: minimizing S also minimizes M and vice-versa.

Why does using Sij give us O(n^5) complexity?

1. If N represents the number of leaves At each stage, we compute S12, S13, S14, …S23, …S(N-1,N), which about N^2 computations.
2. We have N stages (we start off with a matrix of N x N, and at each stage the matrix is reduced by 1) → so we've reached N x N^2 = N^3.
3. Each Sij we compute, requires us to sum over all of the elements in the matrix (see equation 5, last term) – once again, N^2 computations, so now we've reached N x N^2 X N^2 = N^5.

What is Mij?

To define Mij, we first of all define a new parameter called r. This r is computed for each node represented in the current matrix.

$ri = \sum_{k=1}^{N} dik$ (i represents the node for which we are computing r now)

Next, we define a rate corrected matrix (M), in which the elements are defined by:

Mij = dij – (ri+rj) / (N-2) (7)

And this is now our new parameter. Once again – i and j represent numbers of nodes. At each stage, we look for the i and j which give us the minimal Mij.

Why is minimizing Mij equivalent to minimizing Sij?

To prove the equivalence, we'll use the following equation:

$$\sum_{3\le i<j\le N} Dij = \sum_{1\le i<j\le N} Dij - \sum_{1\le i\le N} (D_{1i} + D_{2i}) + D_{12} \qquad (8)$$

(The easiest way to understand this equation is simply to write it out and figure it out yourself…)

Then, using this equation, we can get:

$$S12 = \frac{1}{2} M12 + \frac{1}{N-2} \sum_{1\le i<j\le N} Dij$$

(once again… work it out yourself, it works!)

So… you can see that if we minimize Mij, we also minimize Sij.

Why does Mij give us complexity of O(N^3)?

The main difference between these two parameters is:

In Sij, in each round we have to evaluate $\frac{1}{N-2}\sum_{3\leq i<j} D_{ij}$ ($\rightarrow$ O(N^2), see equation 6) .

On the other hand, in Mij we only have to evaluate ri and rj each round. This can be achieved in O(1), if we compute these terms *once* at the beginning of the round.
Thus, if we return to the list that built the complexity of Sij, we get:
Stage 1 and 2 remain with the same complexity $\rightarrow$ O(N^3). Stage 3 is reduced to O(1), and thus we get a total of O(N^3).

<u>And now, if you want to get straight to the point – here is the algorithm outline:</u>

1. Align sequences (without gaps – any gaps are skipped/ignored).
2. Build distance matrix between the sequences. The distance between two sequences can be defined in many ways. The simplest way to define distance between 2 sequences is the p-distance =
   (number of different positions)/(total number of positions).
   Other methods of defining distance are Jukes-Cantor method, Kimura method.
   An example for a distance matrix is:

Taxa:	Bsu	Bst	Lvi	Amo	Mlu
Bsu	-	.1715	.2147	.3091	.2326
Bst		-	.2991	.3399	.2058
Lvi			-	.2795	.3943
Amo				-	.4289
Mlu					-

- Note that the matrix is symmetric, therefore only values above the diagonal need be computed.

3. Compute the r_i's for each I (for each one of the nodes in our matrix):
   $ri = \sum_{k=1}^{N} dik$ (i represents the node for which we are computing r now)
   (dii = 0 – obvious)

4. Compute Mij (for each i<j, since the matrix is symmetric):
   Mij = dij – (ri+rj) / (N-2)
   Save the minimal i and j for which Mij is minimal (note that you don't need to save all the Mij values...)

5. Defined a new node u whose three branches join i, j and the rest of the tree. Define the lengths of the tree branch from u to i and j:
   $L_{iu} = d_{ij}/2 + (r_i-r_j)/[2(N-2)]$
   $L_{ju} = d_{ij}-L_{iu}$
   These distances are the lengths of the new branches.
   (Note that these are different estimates than those given in the original paper – this is because of the change from Sij to Mij).

6. Define the distance from u to each other node ($k \neq i$ or j) as :
   $d_{ku}=(d_{ik}+d_{jk}-d_{ij})/2$
   (Once again – note this is a different estimate from the one given in the original paper).

7. Remove distances to nodes i and j from the data matrix, and
   decrease N by 1.

8. If more than two nodes remain, go back to step 1. Otherwise, the tree is fully defined except for the length of the branch joining the the two remaining nodes (i and j). Let this remaining branch be: $L_{ij}=d_{ij}$.