GMR Genetics and Molecular Research
Online Journal
ISSN:1676-5680

*Methodology*

# A fast neighbor joining method

**J.F. Li**

School of Computer Science and Technology,
Civil Aviation University of China, Tianjin, China

Corresponding author: J.F. Li
E-mail: jfli@cauc.edu.cn

Genet. Mol. Res. 14 (3): 8733-8743 (2015)
Received March 2, 2015
Accepted April 6, 2015
Published July 31, 2015
DOI http://dx.doi.org/10.4238/2015.July.31.22

**ABSTRACT.** With the rapid development of sequencing technologies, an increasing number of sequences are available for evolutionary tree reconstruction. Although neighbor joining is regarded as the most popular and fastest evolutionary tree reconstruction method [its time complexity is $O(n^3)$, where $n$ is the number of sequences], it is not sufficiently fast to infer evolutionary trees containing more than a few hundred sequences. To increase the speed of neighbor joining, we herein propose *FastNJ*, a fast implementation of neighbor joining, which was motivated by *RNJ* and *FastJoin*, two improved versions of conventional neighbor joining. The main difference between *FastNJ* and conventional neighbor joining is that, in the former, many pairs of nodes selected by the rule used in *RNJ* are joined in each iteration. In theory, the time complexity of *FastNJ* can reach $O(n^2)$ in the best cases. Experimental results show that *FastNJ* yields a significant increase in speed compared to *RNJ* and conventional neighbor joining with a minimal loss of accuracy.

**Key words:** Evolutionary tree reconstruction; Neighbor joining; *FastNJ*

 

## INTRODUCTION

Evolutionary tree reconstruction is a basic and important research field in bioinformatics. A rich variety of evolutionary tree reconstruction methods has been developed. These methods can be divided into three categories: distance-based, maximum parsimony, and maximum likelihood. With the time complexity of $O(n^3)$ (where $n$ is the number of sequences), the neighbor joining distance-based method (Saitou and Nei, 1987) is often regarded as the fastest evolutionary tree reconstruction method. Moreover, owing to the topological accuracy demonstrated in many studies (Mihaescu et al., 2009), neighbor joining has been widely used by molecular biologists.

With the rapid development of sequencing technologies, an increasing number of sequences are available for evolutionary tree reconstruction. For example, there are currently 14,831 families in the Pfam database (Finn et al., 2006), where the number of sequences in approximately 52% of families is more than 1000, while the number of sequences in approximately 7% of families is more than 5000. However, neighbor joining is not sufficiently fast to infer evolutionary trees containing more than a few hundred sequences. The main idea of neighbor joining is to iteratively join the pair of nodes with $\min_{i,j} Q(i,j)$; the most time-intensive aspect of each iteration is searching for the pair of nodes to join. Since 2000, a method of increasing the speed of neighbor joining has become a research focus. Many methods have been proposed to improve neighbor joining by reducing the time spent on finding nodes to join or by reducing iteration times.

Mailund et al. (2006), for example, published a fast neighbor joining approach called *QuickJoin* to speed up the search for $\min_{i,j} Q(i,j)$ by a quad-tree. The quad-tree is built according to an approximated matrix of $Q$, and the nodes of the quad-tree store the information about the lower bounds on parts of the $Q$ matrix. Then, the process of searching in $Q$ for $\min_{i,j} Q(i,j)$ is transformed into a process of searching in the quad-tree. During this search, *QuickJoin* does not spend time in exploring those sub-trees whose lower bounds are higher than the current minimal of $Q$. This avoids the scanning of all $Q(i,j)$ and it gains considerable time savings. *QuickJoin* can construct the same evolutionary trees as canonical neighbor joining. It can reduce the practical running time of neighbor joining to $\Theta(n^2)$; nevertheless, in the worst case, the running time remains $O(n^3)$. Because an additional quad-tree is stored, *QuickJoin* is space-consuming. This makes it infeasible to use *QuickJoin* for reconstructing evolutionary trees that contain more than 8,000 sequences.

Instead of joining pairs of nodes with $\min_{i,j} Q(i,j)$ for all $i$ and $j$, as in conventional neighbor joining, relaxed neighbor joining (*RNJ*) (Evans et al., 2006) joins nodes $i$ and $j$ that meet $Q(i,j) \leq Q(i,k) \wedge Q(i,j) \leq Q(k,j)$ for $0 \leq k < r$ and $k \neq i, k \neq j$. Once such a pair of nodes is found, the procedure of searching for the best pair stops at this point, which avoids the searching of all $Q(i,j)$. The worst case running time for *RNJ* is $O(n^3)$. However, an efficient implementation of *RNJ* called *Clearcut* (Sheneman et al., 2006) shows that *RNJ* is significantly faster in practice than both *QuickJoin* and conventional neighbor joining. There is no guarantee that *RNJ* will join pairs with the minimal value in $Q$; therefore, the trees produced by *RNJ* can significantly differ from those produced by neighbor joining. However, experiments have shown that *RNJ* can reconstruct evolutionary trees with accuracy comparable to that of conventional neighbor joining for additive matrices.

Fast neighbor joining (*FNJ*) (Elias and Lagergren, 2009) is another approach that improves neighbor joining by modifying the selection criterion. The basic idea in *FNJ* is to

maintain a set, $L$, which contains $O(n)$ pairs that are all likely candidates for minimal $Q(i, j)$, and then to search for minimal $Q(i, j)$ only from the pairs in $L$. Because the size of $L$ is always $O(n)$, it takes $O(n)$ time to search for the minimal $Q(i, j)$ in each iteration. After each join, not all entries in $L$ are updated; rather, only the ones relative to $i$ or $j$ are updated. That is, all cluster pairs where $i$ or $j$ is an element are removed from $L$. Next, all $Q$ values for the joined cluster ($a = i \cup j$) are computed and the pair $\{a,k\} = min_k\, Q(a, k)$ is inserted in $L$. By using an update formula to compute $Q(a, k)$, this update of $L$ involves time $O(n)$; therefore, the resulting worst case running time for *FNJ* is $O(n^2)$. However, after the first iteration, $L$ is no longer guaranteed to contain the cluster pair that corresponds to $\min_{i,j} Q(i, j)$; consequently, *FNJ* cannot be expected to correctly construct the trees. Elias and Lagergren (2009) focused more attention on *FNJ* accuracy; therefore, we cannot comment on the speedup in actual application.

*RapidNJ* (Simonsen et al., 2008) reduces the running time of neighbor joining by using two auxiliary matrixes, $S$ and $I$, to find the closest pairs before viewing all entries in $Q$. $S$ contains the distances in $D$, but with each row sorted in increasing order, $I$ maps the ordering in $S$ back to positions in $D$. In each iteration, the maximum, $R_{max} = \max_i R_i$, is first determined, where the time spent on calculating all of $R_i$ is $O(r^2)$, and that used to find $R_{max}$ is $O(r)$. Moreover, $Q_{min}$ is initiated as infinity. Then, *RapidNJ* scans the entries in $Q$ row by row. If $Q(i, I(i,j)) < Q_{min}$, then $Q_{min} = Q(i, I(i, j))$, and the best pair is $\{i, j\}$. However, *RapidNJ* stops searching row $I$ when $S_{ij} - R_i - R_{max} > Q_{min}$ becomes true. Thus, the time used to scan all entries in row $I$ after column $j$ is thereby saved. While the worst-case running time of *RapidNJ* remains $O(n^3)$, experiments on datasets smaller than 10,000 taxa showed that *RapidNJ* outperforms *QuickJoin* and *Clearcut*. Moreover, *RapidNJ* can correctly construct the trees. However, the memory consumption of *RapidNJ* is increased on account of the two additional matrices, $S$ and $I$. Consequently, research efforts have been devoted to reducing the memory consumption of *RapidNJ*, such as *ErapidNJ* (Simonsen et al., 2011) and *NINJA* (Wheeler, 2009).

*FastJoin* (Wang et al., 2012) shows that, in an additive matrix, besides $i_0$ and $j_0$, with the minimal $Q$ value for all $i$ and $j$ being true neighbors, $i'$ and $j'$ with the smallest $Q$ value for all $i (i \neq i_0)$ and $j (j \neq j_0)$ are also true neighbors. Therefore, based on the upper bound computation optimization of *RapidNJ*, and the external storage of *ErapidNJ* methods, *FastJoin* improves neighbor joining by selecting two pairs of nodes and merging them as two new nodes in each iteration. Thus, the number of iterations in *FastJoin* is reduced by half. The time complexity of *FastJoin* remains $O(n^3)$; however, experiments show that *FastJoin* can efficiently improve *RapidNJ*.

Furthermore, with the exponential growth of computing power over the past 10 years, along with the ubiquitous availability of different hardware platforms - such as multi-processor and multi-core computers, computer clusters, and graphics processing units (GPUs) - many parallel algorithms have been proposed to improve neighbor joining. For example, Rucci et al. (2013) presented a parallel algorithm for neighbor joining based on the multicore cluster, Sahoo et al. (2010) proposed a parallel algorithm based on the Pthread library, and Al-Neama et al. (2014) implemented a parallel algorithm on OpenMP. In addition, Du and Feng (2006) proposed the *pNJTree* parallel method for neighbor joining using a message passing interface (MPI) running on a workstation cluster, and Liu et al. (2009) developed a parallel neighbor joining algorithm based on GPUs.

From the above examples, it is evident that the speeding up of neighbor joining has become an important issue in evolutionary tree reconstruction. Motivated by the simplicity and efficiency of *RNJ* and *FastJoin*, we therefore propose *FastNJ*, a fast implementation of

neighbor joining. The main idea of *FastNJ* is that all pairs of nodes $i$ and $j$ that meet $Q(i, j) \leq Q(i, k) \wedge Q(i, j) \leq Q(k, j)$ for $0 \leq k < r$ and $k \neq i$, $k \neq j$ are joined in each iteration. Thus, the total iteration time can be reduced and the total running time can thereby be expected to be decreased.

The remainder of this paper is organized as follows. In the 'Methods' section, we introduce the conventional neighbor joining method. It addition, we detail the process of *FastNJ*, derive the distance update formula, and analyze its time complexity in theory. In the 'Results and Discussion' section, we experimentally evaluate the efficiency of *FastNJ*. In the 'Conclusions' section, we summarize the paper.

## MATERIAL AND METHODS

### Conventional neighbor joining

Neighbor joining is a greedy algorithm that attempts to minimize the sum of all branch-lengths on the constructed tree. Conceptually, it begins with a star-formed tree, whereby each leaf node corresponds to a sequence. It iteratively selects two nodes adjacent to the root and joins them by inserting a new node between the root and the two selected nodes (Guo et al., 2008). When joining nodes, the method selects the pair of nodes $i$ and $j$ that are closest under the transformed distance measure (Equation 1):

$$Q(i,j) = (r-2)d_{ij} - R_i - R_j \qquad \text{(Equation 1)}$$

where $d_{ij}$ is the distance between nodes $i$ and $j$ (which assumes symmetry; that is, $d_{ij} = d_{ji}$), $R_k$ is the sum over row $k$ of the distance matrix $R_k = \Sigma_x d_{kx}$ (where $x$ ranges over all nodes adjacent to the root node), and $r$ is the remaining number of nodes adjacent to the root. Once the pair $i$ and $j$ is selected to join, a new node, $C$, which represents the root of the new cluster, is created. Then, the length of branches $(C, i)$ and $(C, j)$ is computed according to Equation 2 (Guo et al., 2008):

$$d_{Ci} = \frac{1}{2}\left(d_{ij} + \frac{R_i - R_j}{r-2}\right), d_{Cj} = \frac{1}{2}\left(d_{ij} + \frac{R_j - R_i}{r-2}\right) \qquad \text{(Equation 2)}$$

Finally, the "distance matrix $D$ is reduced by replacing the distances relative to sequence $i$ and sequence $j$ by those between the new node $C$ and any other node $k$" (Li and Guo, 2008). Distance $d_{Ck}$ is given by Equation 3:

$$d_{Ck} = \frac{1}{2}\left(d_{ik} - d_{iC}\right) + \frac{1}{2}\left(d_{jk} - d_{jC}\right) \qquad \text{(Equation 3)}$$

 

From the above, we can see that, in each iteration, neighbor joining "uses time $O(r^2)$ to search for $\min_{i,j} Q(i, j)$, and it joins $i$ and $j$, uses time $O(r)$ to update $D$, and there are $n$ - 3 iterations ($n$ is the number of sequences in $D$)" (Guo et al., 2008). Therefore, "the total time complexity becomes $O(n^3)$, and the space complexity becomes $O(n^2)$" (Guo et al., 2008).

## FastNJ

Like conventional neighbor joining, *FastNJ* is used to iteratively join nodes. However, the difference between *FastNJ* and conventional neighbor joining is that, in the latter, only one pair of nodes is joined in each iteration, and the total number of iterations is $n$ - 3. In *FastNJ*, on the other hand, multiple pairs of nodes $i$ and $j$ that meet $Q(i, j) \le Q(i, k) \wedge Q(i, j) \le Q(k, j)$ for $0 \le k < r$ and $k \ne i$, $k \ne j$ are joined in each iteration. Accordingly, the iteration time in *FastNJ* is much shorter than that in conventional neighbor joining. In detail, as shown in Figure 1, *FastNJ* includes the following steps in each iteration.

First, for row $i$ in $D$, it computes $Q(i, j)$ according to (1) for $0 \le j \le r$, stores $min_{i, 0 \le j \le r} Q(i, j)$ in *min* and stores the indexes of minimums in *min_index_vector* [$i$] [1... *min_no*] (Step 2.1).

Second, it finds all pairs of nodes that meet $Q(i, j) \le Q(i, k) \wedge Q(i, j) \le Q(k, j)$ for $0 \le k < r$ and $k \ne i$, $k \ne j$ and stores them in array *nodes_to_join* [1...2 * *num_node_to_join*] (Step 2.4).

Third, it joins the neighbors in *nodes_to_join* and produces *num_node_to_join* new nodes (Step 2.6).

Finally, it updates $D$ according to the following two cases (Steps 2.7 and 2.8):

1) If $i$ is a new node generated in Step 2.6, the distance between $i$ and the other node, $j$, is updated according to (2).

2) If $i$ and $j$ are both new nodes - supposing that it is feasible that $i$ is produced by joining $a$ and $b$, and that $j$ is produced by joining $c$ and $d$ - then the distance between $i$ and $j$ is updated according to Equation 4:

$$d_{ij} = \frac{1}{2}\left(\frac{1}{2}\left(d_{ac} + d_{bc} + d_{ad} + d_{bd}\right) - d_{ab} - d_{cd}\right) \quad \text{(Equation 4)}$$

The derivation process of Equation 4 is shown as follows. According to Equation 2, we can obtain Equation 5:

$$d_{ic} = \frac{1}{2}\left(d_{ac} - d_{ai}\right) + \frac{1}{2}\left(d_{bc} - d_{bi}\right), d_{id} = \frac{1}{2}\left(d_{ad} - d_{ai}\right) + \frac{1}{2}\left(d_{bd} - d_{bi}\right) \quad \text{(Equation 5)}$$

According to Equation 1, we can obtain Equation 6:

$$d_{ia} = \frac{1}{2}\left(d_{ab} + \frac{R_a - R_b}{r-2}\right), d_{ib} = \frac{1}{2}\left(d_{ab} + \frac{R_b - R_a}{r-2}\right) \quad \text{(Equation 6)}$$

According to Equations 5 and 6, we can obtain Equation 7:

$$d_{ic} = \frac{1}{2}\left(d_{ac} - d_{ai}\right) + \frac{1}{2}\left(d_{bc} - d_{bi}\right) \quad \text{(Equation 7)}$$

$$= \frac{1}{2}\left(d_{ac} + d_{bc} - \left(d_{ai} + d_{bi}\right)\right)$$

$$= \frac{1}{2}\left(d_{ac} + d_{bc} - \left(\frac{1}{2}\left(d_{ab} + \frac{R_a - R_b}{r-2}\right) + \frac{1}{2}\left(d_{ab} + \frac{R_b - R_a}{r-2}\right)\right)\right)$$

$$= \frac{1}{2}\left(d_{ac} + d_{bc} - d_{ab}\right)$$

In a similar way, we can obtain Equation 8:

$$d_{id} = \frac{1}{2}\left(d_{ad} + d_{bd} - d_{ab}\right) \quad \text{(Equation 8)}$$

From Equations 7 and 8, we can obtain Equation 9:

$$d_{ij} = \frac{1}{2}\left(d_{ic} + d_{id} - d_{cd}\right) \quad \text{(Equation 9)}$$

$$= \frac{1}{2}\left(\frac{1}{2}\left(d_{ac} + d_{bc} - d_{ab}\right) + \frac{1}{2}\left(d_{ad} + d_{bd} - d_{ab}\right) - d_{cd}\right)$$

$$= \frac{1}{2}\left(\frac{1}{2}\left(d_{ac} + d_{bc} + d_{ad} + d_{bd}\right) - d_{ab} - d_{cd}\right)$$

Then, (4) is achieved.

As shown in Figure 1, the time consumption of *FastNJ* in each iteration is primarily reflected in the following five points:

1) It computes $Q(i,j)$ and finds $min_{i,\,0 \le j \le r} Q(i,j)$ for every row $i$ (Step 2.1). For every row $i$, the time used to compute $Q(i,j)$ is $O(r)$, and the time used to find $min_{i,\,0 \le j \le r} Q(i,j)$ is also $O(r)$. If there are $r$ rows, then the total time consumption is $O(r^2)$.

2) It finds all the pairs of nodes that can be joined (Step 2.4). For node $i$, there are $min\_no_i$ nodes $j$ that meet $Q(i,j) == min_{i,\,0 \le j \le r} Q(i,j)$; for every $j$, there are $min\_no_j$ nodes $k$ that meet $Q(j,k) == min_{i,\,0 \le j \le r} Q(j,k)$. In addition, node $i$ can only be joined with another node, $j$; therefore, once a node $j$ that meets $Q(i,j) \le Q(i,k) \wedge Q(i,j) \le Q(k,j)$ is found, other nodes

after $j$ in *min_index_vector* [$i$] [1... *min_no*] will not be scanned. Therefore, for $i$, *FastNJ* uses $O(min\_no_i * min\_no_j)$ to find node $j$ to be joined with $i$ at the most. If there are $r$ nodes, then the total time consumed in this step is $O(r * min\_no_i * min\_no_j)$. For node $i$, $min\_no_i$ is generally very small relative to $r$; therefore, it can be neglected. The time used in this step is $O(r)$.

```
Algorithm: FastNJ
Input: Distance matrix D=(D_ij)_m*n
Output: Evolutionary tree
1.    r←n;
2.    While(r>2)
2.1    For i=0 to r–1 do
           compute Q(i,j) according to (1) for 0≤j≤r;
           min←min_{i, 0≤j≤r} Q(i, j);
           min_no=1;
         For j=0 to r– 1 do
if(|Q(i,j) –min|<0. 1*10^-9)
min_index_vector[i][min_no]=j;
min_no++;
           min_index_vector[i][0]= min_no – 1;
2.2  flag_joined[0~r]←– 1;
2.3  num_node_to_join←0;
2.4  For ( i=0;i<r;)
         if(flag_joined[i]==– 1)
          for(j=1;j<=min_index_vector[i][0];j++)
          if(min_index_vector[i][j]>i)
          for(k=1;k<=min_index_vector[j][0];k++)
                  if(min_index_vector[min_index_vector[i][j]][k]==i)
                  num_node_to_join++;
                  if (i<min_index_vector[i][j])
                  nodes_to_join[2*num_node_to_join – 1]=i;
                       nodes_to_join[2*num_node_to_join]=min_index_vector[i][j];
                    else
                  nodes_to_join[2*num_node_to_join – 1]=min_index_vector[i][j];
                  nodes_to_join[2*num_node_to_join]=i;
                  flag_joined[i]=1;
                  flag_joined[min_index_vector[i][j]]= 1;
                  goto L;
          L:i++;
2.5  nodes_to_join[0]←num_node_to_join;
2.6  for(j=1;j<=nodes_to_join[0];j+=2)
          a←nodes_to_join[2*j– 1];
          b←nodes_to_join[2*j];
          joina and b;
2.7for(i=0;i<r;i++)//O(rx)
        if(flag_joined[i]== – 1)
          for(j=1;j<=nodes_to_join[0];j++)
          a←nodes_to_join[2*j– 1];
          b←nodes_to_join[2*j];
          D[i,a]=0.5*(D[i,a]+D[i,b] –D[a,b]);
2.8for(j=1;j<nodes_to_join[0];j+=2)
          a←nodes_to_join[2*j – 1];
          b←nodes_to_join[2*j];
          for(k=j+1;k<=nodes_to_join[0];k+=2)
          c←nodes_to_join[2*k– 1];
          d←nodes_to_join[2*k];
           D[a,c]=0.5*(0.5* D[a,c]+0.5* D[a,d]+0.5* D[b,c]+0.5* D[b,d]–D[a,b] –D[c,d];
2.9 for(j=1;j<nodes_to_join[0];j++)
b←nodes_to_join[2*j];
       delete row b from D;
2.10 r←r –num_node_to_join;
3. Return.
```

**Figure 1.** Pseudo-code of FastNJ.

3) The time used to join the neighbors in *nodes_to_join* and to produce *num_node_to_join* new nodes (Step 2.6) is $O(num\_node\_to\_join)$.

4) It then updates the distances between the new nodes and other nodes (Step 2.7). There are *num_node_to_join* new nodes and *r* - *num_node_to_join* old ones; the time used to update the distances between them is $O[(r - num\_node\_to\_join) * num\_node\_to\_join)]$.

5) It updates the distances between the new nodes (Step 2.8). The time consumed is $O(num\_node\_to\_join * num\_node\_to\_join)$.

From the above five points, we can see that, in each iteration, the time consumed is $O(r^2 + r * num\_node\_to\_join)$ and that, in the next iteration, *r* is updated to *r* - *num_node_to_join*. Therefore, the total time consumed with *FastNJ* depends on *num_node_to_join* in each iteration. Moreover, *num_node_to_join* in each iteration ranges from 1 to *r* / 2. When *num_node_to_join* equals 1, the total time consumed with *FastNJ* is $O(n^3)$, which is the same as in conventional neighbor joining. Furthermore, *FastNJ* is reduced to *RNJ*; when *num_node_to_join* equals *r* / 2, then the total time consumed with *FastNJ* is $o(\left(n^2 + n \times \frac{n}{2}\right) + \left(\left(\frac{n}{2}\right)^2 + \frac{n}{2} \times \frac{n}{4}\right) + \left(\left(\frac{n}{4}\right)^2 + \frac{n}{4} \times \frac{n}{8}\right) + \cdots \ldots)$, and the iteration times are $\log_2 n$. Thus, the time complexity of *FastNJ* is between $O(n^2)$ and $O(n^3)$.

## RESULTS AND DISCUSSION

To test the efficiency of *FastNJ*, we performed two experiments that compared *FastNJ* and *Clearcut*, which is an implementation of *RNJ*. All experiments were performed on a personal IBM PC with a 2.0-GHZ CPU and 1 GB of RAM on a Linux system.

In the first experiment, *FastNJ* was compared with *Clearcut* to test the speed of *FastNJ*. In this experiment, the test data were 20 protein sequence alignments in which the number of sequences ranged from 2000 to 12,000 randomly selected from Pfam. The number of sequences in each dataset is shown in Table 1.

**Table 1.** Number of sequences in each dataset.

| | Number of sequences | | Number of sequences | | Number of sequences |
|---|---|---|---|---|---|
| Data1 | 2,289 | Data8 | 5,857 | Data15 | 7,290 |
| Data2 | 2,819 | Data9 | 6,098 | Data16 | 8,344 |
| Data3 | 3,802 | Data10 | 6,213 | Data17 | 8,927 |
| Data4 | 4,271 | data11 | 6,639 | Data18 | 9,521 |
| Data5 | 5,088 | Data12 | 6,649 | Data19 | 10,133 |
| Data6 | 5,216 | Data13 | 6,717 | Data20 | 11,288 |
| Data7 | 5,385 | Data14 | 6,882 | | |

We used the PHYLIP *Protdist* program (Felsenstein, 2014) to estimate the pairwise distances according to the Jones-Taylor-Thornton matrix model. The running time of *Clearcut* and *FastNJ* on each dataset is shown in Table 2. In the table, the first and second columns for each dataset are for the respective running times of *Clearcut* and *FastNJ*. The ratio column presents the ratios between the differences in *Clearcut* and *FastNJ* running times and the running times of *Clearcut*.

From the data in Table 2, we derive the following three points:

(1) On all 20 datasets, *FastNJ* was faster than *Clearcut*.

(2) From data in the ratio column, the speedup ratio of *FastNJ* relative to *Clearcut* varied on different datasets; it depended not on the number of sequences in the datasets, but on the shape of the trees.

(3) The average speedup ratio of *FastNJ* relative to *Clearcut* on all 20 datasets was 26.11%.

 

**Table 2.** Running time of *Clearcut* and *FastNJ* on each dataset.

| | *Clearcut* | *FastNJ* | Ratio (%) | | *Clearcut* | *FastNJ* | Ratio (%) |
|---|---|---|---|---|---|---|---|
| Data1 | 0.677 | 0.65 | 3.99 | Data11 | 9.60 | 8.41 | 12.40 |
| Data2 | 1.61 | 1.46 | 9.32 | Data12 | 12.04 | 4.78 | 60.30 |
| Data3 | 2.8 | 2.14 | 23.57 | Data13 | 6.68 | 4.78 | 28.44 |
| Data4 | 4.33 | 2.72 | 37.18 | Data14 | 11.98 | 10.9 | 9.02 |
| Data5 | 4.28 | 2.59 | 39.49 | Data15 | 8.12 | 6.39 | 21.31 |
| Data6 | 5.77 | 2.49 | 56.85 | Data16 | 15.5 | 15.3 | 1.29 |
| Data7 | 5.35 | 4.26 | 20.37 | Data17 | 18.35 | 16.74 | 8.77 |
| Data8 | 6.99 | 3.56 | 49.06 | Data18 | 22.21 | 17.19 | 22.60 |
| Data9 | 6.49 | 4.198 | 35.32 | Data19 | 21.18 | 20.1 | 5.10 |
| Data10 | 9.42 | 7.9 | 16.14 | Data20 | 36.89 | 14.12 | 61.72 |

In the second experiment, *FastNJ* was compared to *Clearcut* to test the accuracy of *FastNJ* on the simulated data. The test data were produced in the same way as in Desper and Gascuel (2002), which covers the features of most real data sets by choosing parameter values based on random trees. First, 1000 96-sequence model trees were generated using the stochastic speciation process described by Kuhner and Felsenstein (1994). These trees were then made non-ultrametric by multiplying the edge lengths with $1.0 + \mu X$, where $X$ follows the standard exponential distribution and $\mu$ is a tuning factor for adjusting the deviation from the molecular clock. In this experiment, $\mu$ was set to 0.6. Then, we set the mutations per site as 0.02, 0.04, and 0.10 to rescale 1000 trees in order to obtain slow, moderate, and fast evolutionary rates. Subsequently, sequence data were generated according to the Kimura two-parameter model with a transition/transversion ratio of 2.0 using the Seq-Gen program (Rambaut and Grassly, 1997). The sequence length was set to 500 sites. Finally, the PHYLIP *Dnadist* program was used to compute the pairwise distance matrices by assuming the Kimura model with a known transition/transversion ratio.

The accuracy was measured by the Robinson-Foulds (RF) distance (Robinson and Foulds, 1979) between the inferred tree and true tree. This distance corresponds to the proportion of internal branches that are found in one tree and not in another. Its value ranges from 0.0 (both trees are identical) to 1.0 (they do not share a branch in common). Table 3 shows the average RF distance of neighbor joining for *Clearcut* and *FastNJ* of 1000 datasets with various rates of evolution.

**Table 3.** Average RF distance between *Clearcut* and *FastNJ* with various rates of evolution.

| | Slow | Moderate | Fast |
|---|---|---|---|
| Neighbor joining | 0.189 | 0.120 | 0.118 |
| Clearcut | 0.195 | 0.120 | 0.1318 |
| FastNJ | 0.208 | 0.138 | 0.1408 |

From the data in Table 3, we can derive the following three points:

(1) With slow to fast evolution rates, all average RF distances of neighbor joining were greater than 0.0, which means that neighbor joining could not correctly find the true trees. This is consistent with the fact that neighbor joining can reconstruct the correct tree only when the matrix is nearly additive (Atteson, 1999). However, the datasets in this experiment, like most real datasets, were far from being additive.

 

(2) With slow to fast evolution rates, the average RF distances of neighbor joining decreased, which is consistent with previous experimental results (Li, 2009).

(3) With a moderate rate of evolution, the RF distance of *Clearcut* was the same as that of neighbor joining. With slow and fast rates, the RF distance of *Clearcut* was greater than that of neighbor joining. This means that, although it was reported that *Clearcut* can reconstruct evolutionary trees for additive matrices with accuracy comparable to the canonical neighbor joining method, the accuracy of *Clearcut* decreased in real datasets.

(4) With all three rates of evolution, the RF distance of *FastNJ* was greater than that of *Clearcut*. Moreover, compared to the RF distance of *FastNJ*, the average difference between the RF distance of *FastNJ* and that of *Clearcut* was 8.9%. That is, the accuracy of *FastNJ* decreased by 8.9% compared to that of *Clearcut*.

From these experimental results, it is evident that *FastNJ* achieved a significant increase (26.11%) in speed with a minimal (8.9%) decrease in accuracy.

## CONCLUSION

To increase the speed of neighbor joining, we proposed *FastNJ*, a fast implementation of neighbor joining motivated by *RNJ* and *FastJoin*. The primary difference between *FastNJ* and conventional neighbor joining is that, in *FastNJ*, the many pairs of nodes selected by the rule used in *RNJ* are joined in each iteration. In theory, the time complexity of *FastNJ* can reach $O(n^2)$ in the best cases. Experimental results showed that *FastNJ* yields a significant speedup compared to conventional neighbor joining and *RNJ* with a minimal loss in accuracy.

## Conflicts of interest

The authors declare no conflict of interest.

## ACKNOWLEDGMENTS

Research supported by grants from the National Natural Science Foundation of China (#61103005), the Major Program of Tianjin Basic Application and Cutting-Edge Technology Research Plan (#14JCZDJC32500), the Science and Technology Project of Civil Aviation Administration of China (#MHRDZ201206), and the Pre-Research of Major Projects of the Civil Aviation University of China (#3122013P003).

## REFERENCES

Al-Neama MW, Reda NM and Ghaleb FFM (2014). Accelerated guide trees construction for multiple sequence alignment. *Int. J. Adv. Res.* 2: 14-22.

Atteson K (1999). The performance of neighbor-joining methods of phylogenetic reconstruction. *Algorithmica* 25: 251-278.

Desper R and Gascuel O (2002). Fast and accurate phylogeny reconstruction algorithms based on the minimum-evolution principle. *J. Comput. Biol.* 9: 687-705.

Du ZH and Feng B (2006). pNJTree: A parallel program for reconstruction of neighbor-joining tree and its application in ClustalW. *Parallel Comput.* 32: 441-446.

Elias I and Lagergren J (2009). Fast neighbor joining. *Theor. Comput. Sci.* 410: 21-23.

Evans J, Sheneman L and Foster J (2006). Relaxed neighbor joining: A fast distance-based phylogenetic tree construction method. *J. Mol. Evol.* 62: 785-792.

Felsenstein J (2014). PHYLIP Home Page. Available at [http://evolution.genetics.washington.edu/phylip.html]. Accessed

October 28, 2014.
Finn RD, Mistry J, Schuster-Bockler B, Griffiths-Jones S, et al. (2006). Pfam: Clans, web tools and services. *Nucleic Acids Res.* 34: D247-D251.
Guo M-Z, Li J-F and Liu Y (2008). A topological transformation in evolutionary tree search methods based on maximum likelihood combining p-ECR and neighbor joining. *BMC Bioinformatics* 9: S4.
Kuhner MK and Felsenstein J (1994). A simulation comparison of phylogeny algorithms under equal and unequal evolutionary rates. *Mol. Biol. Evol*. 11: 459-468.
Li J (2009). Study of methods of constructing evolutionary trees with DNA sequences (in Chinese). PhD dissertation, Harbin Institute of Technology, Harbin.
Li J and Guo M (2008). A new approach to evolutionary tree reconstruction combining particle swarm optimization with p-ECR. *Int. J. Comput. Intell. Res.* 4: 187-195.
Liu Y, Schmidt C and Maskell DL (2009). Parallel reconstruction of neighbor-joining trees for large multiple sequence alignments using CUDA. Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing: May 23-29, 2009, Rome, 1-8.
Mailund T, Brodal GS, Fagerberg R, Pedersen CNS, et al. (2006). Recrafting the neighbor-joining method. *BMC Bioinformatics* 7:29-36.
Mihaescu R, Levy D and Pachter L (2009). Why neighbor-joining works. *Algorithmica* 54: 1-24.
Rambaut A and Grassly NC (1997). Seq-gen: An application for the Monte Carlo simulation of DNA sequence evolution along phylogenetic trees. *Comp. Appl. Biosci.* 13: 235-238.
Robinson D and Foulds L (1979). Comparison of weighted labelled trees. *Lect. Notes Math.* 748: 119-126.
Rucci E, Chichizola F, Naiouf M and De Giusti A (2013). A hybrid parallel neighbor-joining algorithm for phylogenetic tree reconstruction on a multicore cluster. *Parallel Cloud Comput.* 2: 74-80.
Sahoo B, Behura A and Padhy S (2010). Fine grain construction of neighbor-joining phylogenetic trees with reduced redundancy using multithreading. *Int. J. Distributed Parallel Systems* 1: 129-140.
Saitou N and Nei M (1987). The neighbor-joining method: A new method for reconstructing phylogenetic trees. *Mol. Biol. Evol*. 4: 406-425.
Sheneman L, Evans J and Foster JA (2006). Clearcut: a fast implementation of relaxed neighbor joining. *Bioinformatics* 22: 2823-2824.
Simonsen M, Mailund T and Pedersen CNS (2008). Rapid neighbor-joining. Proceedings of the Eighth International Workshop on Algorithms in Bioinformatics: September 15-19, 2008 (Crandall KA and Lagergren J, eds.). Springer Berlin, Heidelberg, Karlsruhe, 113-122.
Simonsen M, Mailund T and Pedersen CNS (2011). Inference of large phylogenies using neighbour-joining. *Comm. Comp. Inf. Sci.* 127: 334-344.
Wang J, Guo M-Z and Xing LL (2012). FastJoin, an improved neighbour-joining algorithm. *Genet. Mol. Res.* 11: 1909-1922.
Wheeler TJ (2009). Large-scale neighbor-joining with NINJA. *Lect. Notes Comput. Sci.* 5724: 375-389.