Rapid Neighbour-Joining

Martin Simonsen, Thomas Mailund and Christian N. S. Pedersen

Bioinformatics Research Center (BIRC), University of Aarhus,
C. F. Møllers Allé, Building 1110, DK-8000 Århus C, Denmark.
{kejseren,mailund,cstorm}@birc.au.dk

Abstract. The neighbour-joining method reconstructs phylogenies by iteratively joining pairs of nodes until a single node remains. The criterion for which pair of nodes to merge is based on both the distance between the pair and the average distance to the rest of the nodes. In this paper, we present a new search strategy for the optimisation criteria used for selecting the next pair to merge and we show empirically that the new search strategy is superior to other state-of-the-art neighbour-joining implementations.

1 Introduction

The neighbour-joining (NJ) method by Saitou and Nei [8] is a widely used method for phylogenetic reconstruction, made popular by a combination of computational efficiency combined with reasonable accuracy. With its cubic running time by Studier and Kepler [11], the method scales to hundreds of species, and while it is usually possible to infer phylogenies with thousands of species, tens or hundreds of thousands of species is infeasible. Various approaches have been taken to improve the running time for neighbour-joining. QuickTree [5] was an attempt to improve performance by making an efficient implementation. It outperformed the existing implementations but still performs as a $O\left(n^3\right)$ time algorithm. QuickJoin [6, 7], instead, uses an advanced search heuristic when searching for the pair of nodes to join. Where the straight-forward search takes time $O\left(n^2\right)$ per join, QuickJoin on average reduces this to $O\left(n\right)$ and can reconstruct large trees in a small fraction of the time needed by QuickTree. The worst-case time complexity, however, remains $O\left(n^3\right)$, and due to a rather large overhead QuickJoin cannot compete with QuickTree on small data sets. Other approaches, such as "relaxed neighbour-joining" [3, 10] and "fast neighbour-joining" [1] modify the optimisation criteria used when selecting pairs to join. The method "relaxed neighbour-joining" has a worst-case $O\left(n^3\right)$ running time while "fast neighbour-joining" has $O\left(n^2\right)$ running time.

In this paper we introduce a new algorithm, RapidNJ, to lower the computing time of canonical neighbour-joining. We improve the performance by speeding up the search for the pair of nodes to join, while still using the same optimisation criteria as the original neighbour-joining method. Worst-case running time remains $O\left(n^3\right)$, but we present experiments showing that our algorithm outperforms both QuickTree, QuickJoin and an implementation of relaxed neighbour-joining on all input sizes.

2 Canonical Neighbour-Joining

Neighbour-joining [8, 11] is a hierarchical clustering algorithm. It takes a distance matrix D as input, where $D(i,j)$ is the distance between cluster i and j. It then iteratively joins clusters by using a greedy algorithm, which minimises the total sum of branch lengths in the reconstructed tree. Basically the algorithm uses n iterations, where two clusters (i,j) are selected and joined into a new cluster. The two clusters are selected by minimising

$$Q(i,j) = D(i,j) - u(i) - u(j)\,, \tag{1}$$

where

$$u(l) = \sum_{k=0}^{r-1} D(l,k)/(r-2)\,, \tag{2}$$

and r is the number of remaining clusters. When the minimum Q-value $q_{\min} = \min_{0 \le i,j < r} Q(i,j)$ is found, D is updated, by removing the i'th and j'th row and column. A new row and column are inserted with the distances of the new cluster. Distance between the new cluster $a = i \cup j$ and an old cluster k, are calculated as

$$D(a,k) = \frac{D(i,k) + D(j,k) - D(i,j)}{2}\,. \tag{3}$$

The result of the algorithm is a unrooted bifurcating tree where the initial clusters corresponds to leafs and each join corresponds to an internal node in the tree.

3 Rapid Neighbour-Joining

We seek to improve the performance of canonical neighbour-joining by speeding up the search for the pair of nodes to join, while still using the same optimisation criteria as the original neighbour-joining method. The overall aim is thus similar to that of QuickJoin, but the approach is different. The RapidNJ algorithm presented in this paper is based on the following observation.

- When searching for $q_{\min}$ in (1), $u(i)$ is constant in the context of row i.

This observation can be used to create a simple upper bound on the values of each row in Q, thereby reducing the search space significantly. The upper bound is dynamically updated based on the Q-values searched. While the algorithm still has a worst-case time complexity of $O\left(n^3\right)$, our experiments, reported later, show that in practise the algorithm performs much better. To utilize the upper bound two, new data structures, S and I, are used. S is a sorted representation of D and I maps S to D. Memory consumption is increased due to S and I. The increase depends on implementation choices, and can be minimized at the cost of speed. Worst case memory consumption remains $O\left(n^2\right)$.

3.1 Data Structures

The two new data structures, S and I, needed to utilize our upper bound, are constructed as follows. Matrix S contains the distances from D but with each row sorted in increasing order. Matrix I maps the ordering in S back to positions in D. Let $o_1, o_2, \ldots, o_n$ be a permutation of $1, 2, \ldots, n$ such that $D(i, o_1) \leq D(i, o_2) \leq \cdots \leq D(i, o_n)$, then

$$S(i,j) = D(i, o_j)\,, \tag{4}$$

and

$$I(i, o_j) = j\,. \tag{5}$$

3.2 Search Heuristic

Matrix S is used for a bounded search of $q_{\min}$. First the maximum u-value $u_{\max}$ needs to be found. Recalculating all u-values, and finding the $u_{\max}$ can be done in time $O(n)$. The following algorithm can then be used to search for the pair (i,j) minimising $Q(i,j)$:

1. Set $q_{\min} = \infty$, $i = -1$, $j = -1$
2. for each row r in S and column c in r:
   (a) if $S(r,c) - u(r) - u_{\max} > q_{\min}$ then move to the next row.
   (b) if $Q(r, I(r,c)) < q_{\min}$ then set $q_{\min} = Q(r, I(r,c))$, $i = r$ and $j = I(r,c)$.

The algorithm searches each row in S and stops searching a row when the condition

$$S(r,c) - u(r) - u_{\max} > q_{\min} \tag{6}$$

becomes true or the end of a row is reached. If we reached an entry in a row where (6) is true, we are looking at a pair (i,j), where $D(i,j)$ is too large for (i,j) to be a candidate for $q_{\min}$, and the following entries in the row $S(i)$, can be disregarded in the search. This is easily seen by remembering that $u(i)$ is constant in context of a row, $u(j) = u_{\max}$ in (6) and $S(r,k) \geq S(r,l)$ when $k > l$. Whenever we see a new $Q(r, I(r,c))$ smaller than the previously smallest, we remember the entry, using I to obtain the right column index in D. Q is never fully calculated as this would require $O\left(n^2\right)$ time in each iteration. We only calculate entries as they are needed.

The number of entries searched in each row depends on the current value of the $q_{\min}$ variable. If $q_{\min}$ is close to the actual minimum value, more entries in a row can most likely be excluded from the search. To improve the overall performance, it is therefore important quickly to find a value for $q_{\min}$ as close as possible to the actual minimum. We propose two strategies for this:

- The first entry in each row can be searched for a good minimum. Intuitively $S(r,0)$ is likely to have a small Q-value compared to entries in the rest of row i.

– Cache one or more rows containing good minimums in each iteration, and use these rows as a primer in the next iteration. The u-values do not change very much in each iteration, so a row with a small q value is likely to have a small q value in the succeeding iteration. Quite often, the minimum q value can be found in the row containing the next best minimum found in the previous iteration.

Since we can risk searching all entries in every row, the worst-case time complexity of the algorithm is $O\left(n^3\right)$.

3.3 Updating the Data Structures

The D matrix is updated as in the original neighbour-joining algorithm. Two rows and two columns, corresponding to the two clusters we merged, are removed. One row and column are inserted corresponding to the new cluster.

The S and I matrices need to be updated to reflect the changes made in the D matrix. Removing the two rows can easily be done, but because the rows of S are sorted, removing the columns is more complicated. Identifying and removing entries in S corresponding to the two columns removed in D, would require at least $O(n)$ time depending on implementation of S. By keeping a record of deleted columns, we can simply leave the obsolete entries in S. Identifying obsolete entries while searching S gives a small performance penalty. The penalty can be minimised by garbage collecting obsolete entries in S, when they take up too much space. Our experiments have shown that most rows in S are removed, before obsolete entries become a problem.

Let d be the number of remaining clusters after merging the cluster-pair (i, j), into a new cluster. A new row with length $d-1$, and sorted by increasing distance is inserted in S. The new row contains all new distances introduced by merging (i, j). A row which maps the new row in S to the corresponding row in D is created and inserted in I. Due to the time required for sorting, creating and inserting the two rows in S and I takes $O(d \log d)$ time.

4 Results and Discussion

To evaluate the performance of our search heuristic, we have implemented the RapidNJ algorithm using the second of the two strategies mentioned above to initialise $q_{\min}$ with a good value. This strategy can be implemented with a small overhead and it often initialises $q_{\min}$ with the actual minimum q value. RapidNJ only caches the row containing the q value closest to $q_{\min}$, where the row is not one of the two rows which are associated with $q_{\min}$. This row is searched before any other row thereby initialising $q_{\min}$. The current implementation of RapidNJ is available at http://www.birc.au.dk/Software/RapidNJ/.

QuickJoin [7], QuickTree [5] and Clearcut [10] have been used as reference in our experiments. We have not been able to locate an implementation of

FastNJ [1] to include in our experiments. Clearcut is an implementation of relaxed neighbour-joining, while QuickJoin and QuickTree implements the canonical NJ method. QuickJoin and QuickTree are both fast implementations and use different heuristics to optimize the running time of the canonical neighbour joining method. Clearcut uses a different algorithm, but it can reconstruct trees with almost the same accuracy as the canonical NJ, and is a good alternative to NJ. Phylip formatted distance matrices were used as input, and Newick formatted phylogenetic trees as output.

4.1 Environment

All experiments were preformed on a machine with the following specifications:

– Intel Core 2 6600 2.4 GHz with 4 MB cache
– 2 GB RAM
– Fedora 6, kernel 2.6.22.9-61 OS

QuickTree and Clearcut were written in *C* while QuickJoin and RapidNJ were written in *C++*. All reference tools were compiled with the `make` script included in the source code. We used the standard `time` tool available in the Fedora 6 distribution for measurements of running time. The "real time" output of the `time` tool was used in all measurements. Three runs of each program on every input was made, and the best time of the three runs was used in order to avoid disturbance from other processes on the system.

4.2 Data

All four implementations were given distance matrices in phylip format as input. We used the following two sources of data to evaluate the performance of the four implementations.

The first data source was protein sequence alignments from Pfam [4]. The alignments were translated into phylip formatted distance matrices using QuickTree. Data sets from Pfam are real data, and we found that most data sets contained lots of redundant data, i.e. the distance matrix contains rows i and k where $\forall j : D(i,j) = D(k,j)$. We also found several data sets where the same taxa was represented twice. Apart from representing real data, Pfam data sets also test how resilient the different optimisations and algorithms used in the four implementations, are to irregular inputs. Figures 1, 2 and 3 show the results of the experiments on these data.

The second data source was based on simulated phylogenetic trees. The trees were generated using r8s [9], which is a tool for simulating molecular evolution on phylogenetic trees. As a feature r8s can also simulate random phylogenetic trees. We used the yule-c model to generate the data used in the experiments. Simulated data sets provide a good basis for testing the algorithmic complexity of the four implementations. They contain no redundant data, and the distance matrices are always additive. Since the trees generated by r8s using the yule-c

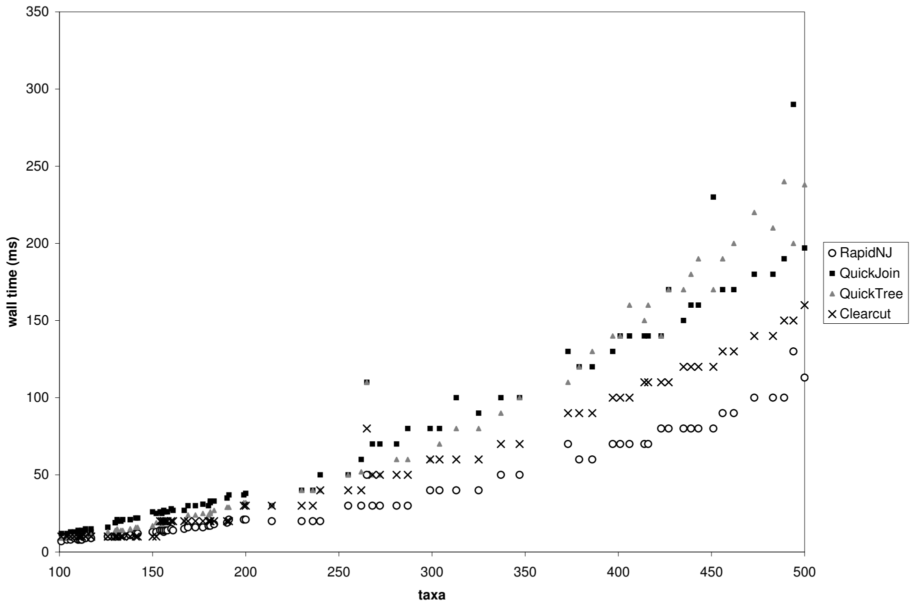

Fig. 1. Performance of RapidNJ compared to QuickJoin, QuickTree and Clearcut on small Pfam data (< 500 taxa).

model are clocklike, the distance matrices are actually ultrametric. This, however, should not yield an advantage for any of the methods, so in this study, we do not make any effort to e.g. perturbe the branch lengths for obtaining non-ultrametric but additive distance matrices. Figure 5 shows the result of the experiments on simulated data sets.

Experiments on data sets larger than 11000 taxa were not performed due to the memory requirements of RapidNJ and QuickJoin. Both have $O\left(n^2\right)$ memory consumption, but with larger constants than Clearcut and QuickTree.

4.3 Results on Pfam Data

On small inputs, see Fig. 1, all four implementations have good performance, only minor differences separates the implementations. QuickJoin does suffer from a large overhead, and has the longest running time, while RapidNJ has the shortest. When we look at medium sized data sets (Fig. 2), QuickJoin starts to benefit from the heuristics used. Again we observe that RapidNJ has a consistently short running time compared to both QuickTree and QuickJoin. Clearcut comes quite close to RapidNJs running time, and only a small constant factor separates the two. QuickTree begin to suffer from the $O\left(n^3\right)$ time complexity on these input sizes, while the other three implementations have a much lower running time. Looking at large input sizes (Fig. 3), QuickTree clearly suffers from its $O\left(n^3\right)$ time complexity, while Clearcut, QuickJoin and RapidNJ continue to perform

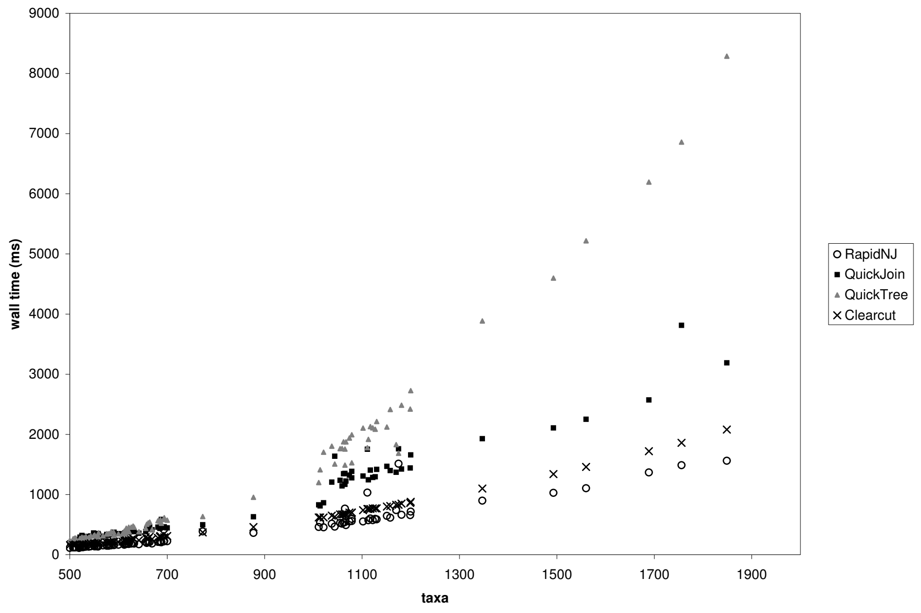

Fig. 2. Performance of RapidNJ compared to QuickJoin, QuickTree and Clearcut on medium Pfam data (500–2000 taxa).

much better. On most data sets RapidNJ has half the running time of QuickJoin, while the running times of Clearcut is nearly identical to those of RapidNJ on most data sets.

Some outliers belonging to all four implementations can be observed in Fig. 1, 2 and 3. The most noticeable are seen on medium and large data sets, where some running times belonging to RapidNJ are up to three times as long compared other data sets of same size. We have investigated the problem, and found that most outliers can be explained by redundant data. Redundant data are taxa with equal distances to all other taxa and a mutual distance of 0. They are quite common in Pfam data sets and affects the performance of RapidNJ negatively. In data sets where the mutual distance between taxa varies, the number of new entries in the Q matrix which fall below the upper bound is roughly the same as the entries which are joined. In data sets containing m redundant taxas the number of q values falling below the upper bound can suddenly explode. Every time an entry in the Q matrix related to a redundant taxa fall below the upper bound, at least $m - 1$ additional entries also falls bellow the upper bound. If m is large enough, the running time of RapidNJ increases significantly. Some alignments from Pfam contain lots of identical sequences which gives redundant data.

Among the deviating data sets seen in Fig. 3 we found some, where over one quarter of all taxa where redundant. QuickJoin is also affected by redundant data but to a less extent than RapidNJ. Clearcut seems to be unaffected while

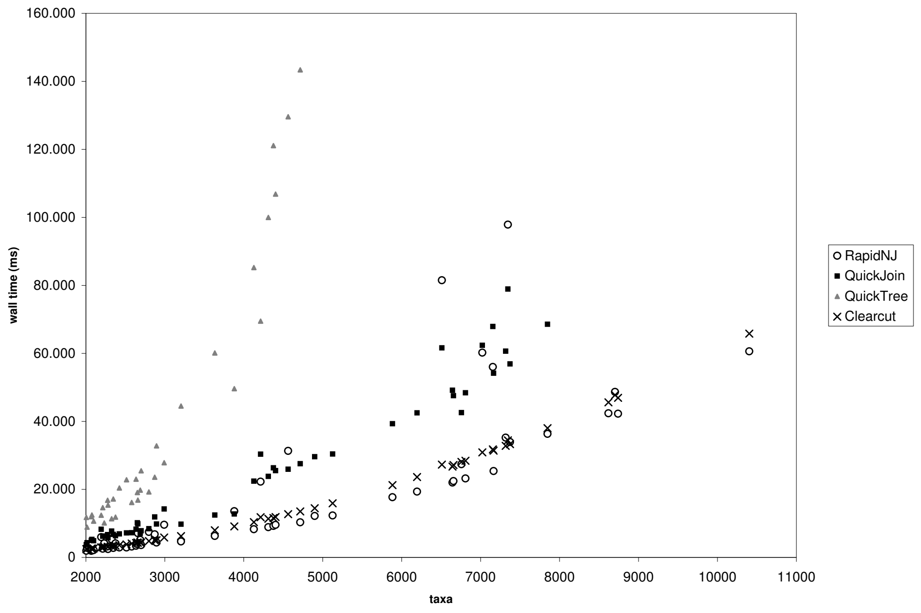

Fig. 3. Performance of RapidNJ compared to QuickJoin, QuickTree and Clearcut on large Pfam data (> 2000 taxa).

QuickTree actually benefits from redundant data due to a heuristic which treats redundant taxa as a single taxon.

Not all deviating running times can be explained by redundant data. We found that a very few data sets from Pfam, contained large sets of taxa where the mutual distances are concentrated in small range. In rare cases these data sets increased the running time of RapidNJ by up to a factor two.

To test the impact of redundant data, we implement a preprocessing phase where identical rows are joined before neighbour-joining is applied. When using this preprocessing phase, the output is a tree with a slightly different topology than those produces by the canonical neighbour-joining algorithm. The difference lies mainly in the subtrees containing redundant data, and can be explained by the order which the redundant taxa were joined. It is a close approximation of a NJ-tree, but this implementation mainly serves to illustrate the problem with redundant data. The result of experiments with preprocessing is presented in Fig. 4, which shows that the number of deviating running times are reduced significantly.

4.4 Results on Simulated Data

Results on simulated data show the same tendencies as the results on Pfam data. These data sets contain no redundant rows, and we see no deviating running times. A clear trend line of all tree implementations can be observed on

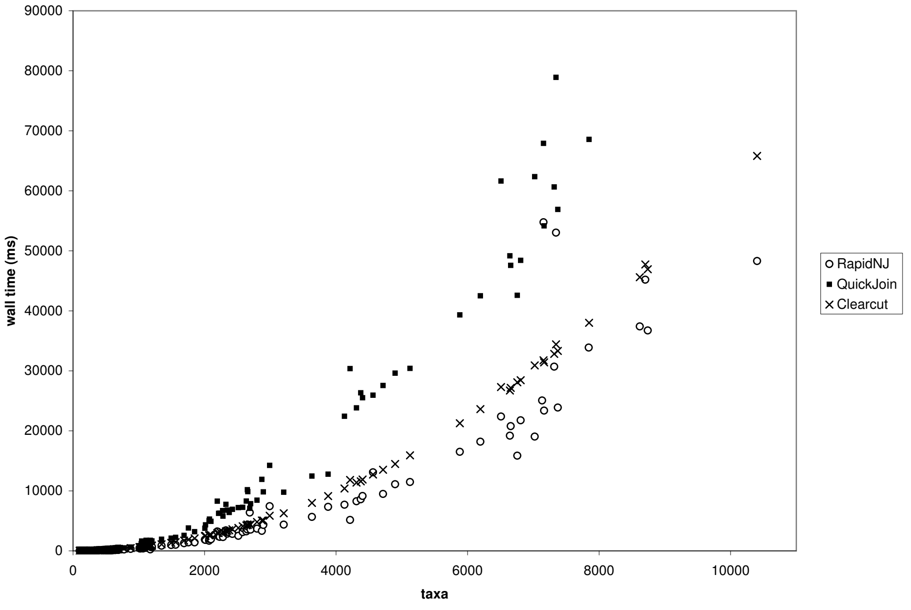

Fig. 4. Performance of RapidNJ with the preprocessing phase where redundant data is eliminated compared to QuickJoin and Clearcut on Pfam data sets.

Fig. 5. It seems Clearcut, QuickJoin and RapidNJ have almost the same asymptotic time complexity, but with different constants. We can observe a number of jumps on QuickJoins curve. These can be explained by the way data structures are implemented in QuickJoin. Every time the input size reaches the next power of two, memory consumption increases by a factor four. This affects the running time thus creating the jumps seen in Fig. 5. The memory consumption of RapidNJ, Clearcut and QuickTree is a direct function of the input size, and we see no sudden jumps in the running times of these implementations.

5 Conclusion

We have presented RapidNJ, a search heuristic used to speed up the search for pairs to be joined in the neighbour-joining method. The search heuristic searches for the same optimisation criteria as the original neighbour-joining method, but improves on the running time by eliminating parts of the search space which cannot contain the optimal node pair. While the worst-case running time of RapidNJ remains $O\left(n^3\right)$, it outperforms state of the art neighbour-joining and relaxed neighbour-joining implementations such as QuickTree, QuickJoin and Clearcut. Since the distance matrix used as input to NJ methods is typically derived from a multiple alignment, it would be interesting to investigate the overall performance of RapidNJ when combined with efficent methods such as [2] to obtain the distance matrix from a multiple alignment.

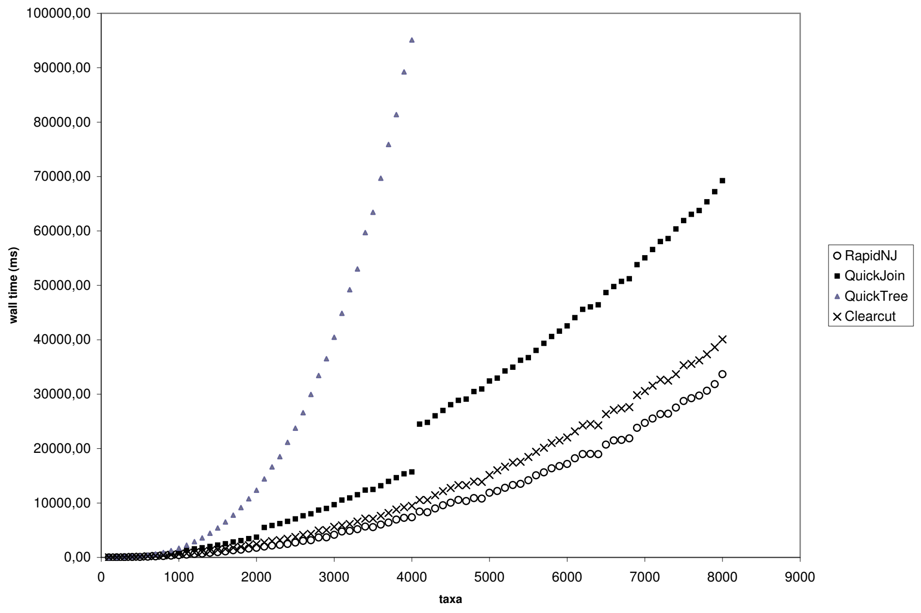

Fig. 5. Performance of RapidNJ compared to QuickJoin, QuickTree and Clearcut on simulated data sets.

References

1. I. Elias and J. Lagergren. Fast neighbour joining. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3580 of *Lecture Notes in Computer Science*, pages 1263–1274. Springer, 2005.
2. I. Elias and J. Lagergren. Fast computation of distance estimators. *BMC Bioinformatics*, 8:89, 2007.
3. J. Evans, L. Sheneman, and J. A. Foster. Relaxed neighbor joining: A fast distance-based phylogenetic tree construction method. *Journal of Molecular Evolution*, 62(6):785–792, 2006.
4. R. D. Finn, J. Mistry, B. Schuster-Böckler, S. Griffiths-Jones, V. Hollich, T. Lassmann, S. Moxon, M. Marshall, A. Khanna, R. Durbin, S. R. Eddy, E. L. L. Sonnhammer, and A. Bateman. Pfam: Clans, web tools and services. *Nucleic Acids Research*, Database Issue 34:D247–D251, 2006.
5. K. Howe, A. Bateman, and R. Durbin. QuickTree: Building huge neighbour-joining trees of protein sequences. *Bioinformatics*, 18(11):1546–1547, 2002.
6. T. Mailund, G. S. Brodal, R. Fagerberg, C. N. S. Pedersen, and D. Philips. Recrafting the neighbor-joining method. *BMC Bioinformatics*, 7(29), 2006.
7. T. Mailund and C. N. S. Pedersen. QuickJoin – fast neighbour-joining tree reconstruction. *Bioinformatics*, 20:3261–3262, 2004.
8. N. Saitou and M. Nei. The neighbor-joining method: A new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4:406–425, 1987.
9. M. L. Sanderson. Inferring absolute rates of molecular evolution and divergence times in the absence of molecular clock. *Bioinformatics*, 19:301–302, 2003.

10. L. Sheneman, J. Evans, and J. A. Foster. Clearcut: A fast implementation of relaxed neighbor-joining. *Bioinformatics*, 22(22):2823–2824, 2006.
11. J. A. Studier and K. J. Kepler. A note on the neighbour-joining method of Saitou and Nei. *Molecular Biology and Evolution*, 5:729–731, 1988.