Alcom-FT Technical Report Series
ALCOMFT-TR-03-102

Speeding Up Neighbour-Joining Tree Construction

Gerth Stølting Brodal[*,†]
BRICS[‡], Department of Compupter Science
University of Aarhus, Denmark
gerth@brics.dk

Rolf Fagerberg[*]
BRICS[‡], Department of Computer Science
University of Aarhus, Denmark
rolf@brics.dk

Thomas Mailund
Bioinformatics Research Center (BiRC)
University of Aarhus, Denmark
mailund@birc.dk

Christian N. S. Pedersen
Bioinformatics Research Center (BiRC)
University of Aarhus, Denmark
cstorm@birc.dk

Derek Phillips
School of Computer Science
University of Waterloo, Canada
djphilli@uwaterloo.ca

ABSTRACT
A widely used method for constructing phylogenetic trees is the neighbour-joining method of Saitou and Nei. We develope heuristics for speeding up the neighbour-joining method which generate the same phylogenetic trees as the original method. All heuristics are based on using a quad-tree to guide the search for the next pair of nodes to join, but differ in the information stored in quad-tree nodes, the way the search is performed, and in the way the quad-tree is updated after a join.

We empirically evaluate the performance of the heuristics on distance matrices obtained from the Pfam collection of alignments, and compare the running time with that of the QuickTree tool, a well-known and widely used implementation of the standard neighbour-joining method. The results show that the presented heuristics can give a significant speed-up over the standard neighbour-joining method, already for medium sized instances.

Categories and Subject Descriptors
F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; J.3 [**Life and Medical Sciences**]: Biology and Genetics

[*]Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).
[†]Supported by the Carlsberg Foundation (contract number ANS-0257/20).
[‡]Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation.

General Terms
Algorithms, Experimentation, Performance

Keywords
Neighbour-joining, tree construction

1. INTRODUCTION
The neighbour-joining method is a distance based method for constructing evolutionary trees. It was introduced by Saitou and Nei [10], and the running time was later improved by Studier and Keppler [12]. It has become a mainstay of phylogeny reconstruction, and is probably the most widely used distance based algorithm in practice. With a running time of $O(n^3)$ on n taxa [12], it is fast, and empirical work shows it to be quite accurate, at least for cases where the rate of evolution is not extremely high or low. St. John et al. [11] even suggest it as a standard against which new phylogenetic methods should be evaluated.

The neighbour-joining method is a greedy algorithm which attempts to minimize the sum of all branch-lengths on the constructed tree. Conceptually, it starts out with a star-formed tree leaf corresponds to a species, and iteratively picks two nodes adjacent to the root and joins them, by inserting a new node between the root and the two selected nodes. When joining nodes, the method selects the pair of nodes i, j that minimizes the branch-length sum of the resulting new tree.

One way of achieving this [12] is always to select the pair of nodes i, j that minimizes

$$Q_{ij} = (r-2)d_{ij} - (R_i + R_j) \ , \tag{1}$$

where d_{ij} is the *distance* between node i and j (assumed symmetric, i.e., $d_{ij} = d_{ji}$), R_k is the *row sum* over row k of the distance matrix: $R_k = \sum_i d_{ik}$ (where i ranges over all nodes adjacent to the root node), and r is the *remaining* number of nodes adjacent to the root. When nodes i and j are joined, they are replaced with a new node, A, with distance to the remaining nodes given by

$$d_{Ak} = (d_{ik} + d_{jk} - d_{ij})/2 \ . \tag{2}$$

The method performs a search for $\min_{i,j} Q_{ij}$, using time $O(r^2)$, and joins i and j, using time $O(r)$ to update d. This search and join is continued until only three nodes are adjacent to the root (i.e. for $n-3$ joins where n is the total number of species), giving a total time complexity of $O(n^3)$, and a space complexity of $O(n^2)$ (for representing the distance matrix d). For further discussions of the neighbour-joining method, see e.g. [5, 8, 9].

In this paper, we present several heuristics for speeding up the neighbour-joining method. The heuristics all attempt to reduce the search time for $\min_{i,j} Q_{ij}$ by using a quad-tree [3] built on top of the Q matrix, or on a matrix related to the Q matrix that does not need to be recomputed after each join. The nodes of the quadtree store information guiding the search for the minimum, and the crux of the heuristics is to define this information in a way which will guide the search well for many iterations before it needs updating.

The time complexity of the heuristics are given by

$$O(nS + U) \ ,$$

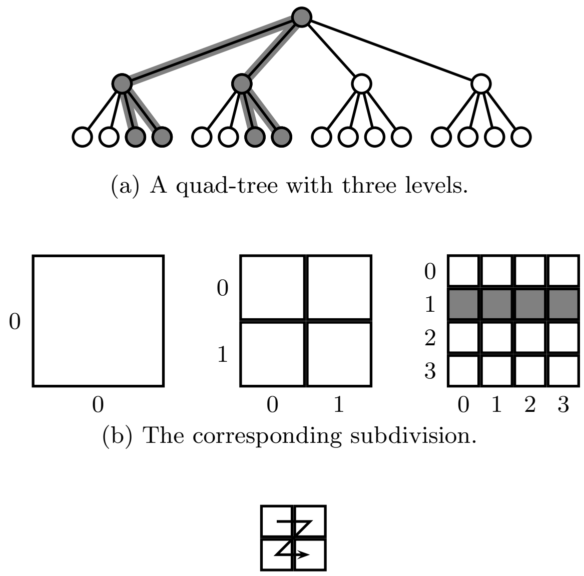
where S is the average *search time* for finding nodes i and j minimizing Q_{ij}, and U is the time used, throughout the algorithm, for *updating* the quad-tree and other bookkeeping information, e.g., the distance matrix. The worst case time complexity remains $O(n^3)$, but the anticipation is that the developed heuresitcs on real data is significantly faster. The space complexity after adding the quad-tree is still $O(n^2)$ since a quad-tree with n^2 leaves can be represented in $O(n^2)$ space.

We evaluate the performance of the heuristics empirically on distance matrices obtained from the Pfam collection of alignments [2, 1], and compare the running time with that of the QuickTree tool [6], a well-know and widely used implementation of the standard neighbour-joining method. The results show that the presented heuristics can give a significant speed-up over the standard neighbour-joining method, already for moderately sized instances. Indeed, evidence is given that the running time of the best of our heuristic evolves as $\Theta(n^2)$ on the examined instance collection, as opposed to $\Theta(n^3)$ for QuickTree.

2. USING A QUAD-TREE

A quad-tree [3] is a four-ary tree modelling a two-dimensional area recursively divided into quadrants. In the following description we assume for the sack of simplicity that r is a power of two. Figure 1 shows the tree resulting from a three-level recursive process.

By building a quad-tree of height $\log r$ on top of the $r \times r$ matrix Q and storing in nodes of the quad-tree the minimal values in the subtree rooted at that node, we can search for the pair of nodes minimizing Q_{ij} in time $O(\log r)$. However by definition (1), in each iteration of the algorithm, all entries in Q need to be updated: the value r is decreased by one, and each row and column in d has a new distance to the joined node A added and two distances removed, thereby changing



(a) A quad-tree with three levels.

(b) The corresponding subdivision.

(c) One possible ordering of child nodes.

Figure 1: A quad-tree with three levels of nodes, and the corresponding subdivision of a square. If the ordering of children is chosen as shown, the highlighted row corresponds to the highlighted leaves.

R_k for all k. Updating Q after each iteration therefore takes time $O(r^2)$, leading to a running time of $O(n^3)$. There is no asymptotic gain, and in practice the quad-tree solution will be significantly slower than the basic, non-quad-tree, algorithm, as a consequence of the added overhead. Simply building a quad-tree on top of Q will not improve the running time.

The problem with building the quad-tree on top of Q is that all entries in Q change with each join. To decrease the update time, we need to build the quad-tree on some information that does not completely change with each join. If, for instance, we only need to update a single row and column per join, we can do that in $O(r)$ time time, as illustrated by Figure 1.

2.1 Using Approximations of Q

If we assume that the relative differences between the Q_{ij} values does not change dramatically between joins—that is, we assume that the ordering of Q_{ij} values is not randomly permuted after a join—we would expect that we could use the old Q_{ij} values to guide the search for the current minimal Q_{ij}. Let Q' denote the Q matrix at some earlier point, and let r' denote the number of remaining nodes adjacent to the root at that point. Similarly, let R'_k denote the row sum of row k in Q', and let δ_k denote the difference between R_k and R'_k: $R_k = R'_k + \delta_k$. Based on these definitions we can rewrite (1) to the following:

$$Q_{ij} = \frac{r-2}{r'-2} Q'_{ij} + \frac{r-r'}{r'-2}(R'_i + R'_j) - (\delta_i + \delta_j) \ . \tag{3}$$

This equation expresses the current Q_{ij} values in terms of the old values and some *correction terms*, given by the R' and δ vectors. Because of these terms, the minimal Q'_{ij} does not necessarily identify the nodes i, j that minimizes Q_{ij}, so we cannot use a quad-tree of Q' alone to find the nodes to join. We can, however, use a quad-tree over Q' to get *lower bounds* for the minimal Q_{ij} value in parts of the Q matrix, as described in the following.

Let $\mathcal{Q}$ denote a quad-tree built on top of Q' such that $\mathcal{Q}[i,j,l]$ denotes the minimum value at level l, where leaves are at level zero. More precisely, let $\mathcal{Q}[i,j,0] = Q'_{ij}$, and

$$\mathcal{Q}[i,j,l] \quad = \quad \min \begin{cases} \mathcal{Q}[2i,2j,l-1] \\ \mathcal{Q}[2i+1,2j,l-1] \\ \mathcal{Q}[2i,2j+1,l-1] \\ \mathcal{Q}[2i+1,2j+1,l-1] \ . \end{cases}$$

With this definition, we have

$$\mathcal{Q}[i,j,l] = \min_{2^l i \le i' < 2^l(i+1),\ 2^l j \le j' < 2^l(j+1)} Q'_{i'j'} \ .$$

Let $\mathcal{B}$ denote a binary tree for the correction terms built as follows, where $\mathcal{B}[k,l]$ denotes the kth node at level l:

$$\begin{aligned} \mathcal{B}[k,0] &= \frac{r-r'}{r'-2} R'_k - \delta_k \\ \mathcal{B}[k,l] &= \min\{\mathcal{B}[2k,l-1], \mathcal{B}[2k+1,l-1]\} \ . \end{aligned}$$

We have

$$\mathcal{B}[k,l] \quad = \quad \min_{2^l k \le k' < 2^l(k+1)} \mathcal{B}[k',0] \ .$$

From the rewritting of Q by (3) and the trees above, we define

$$\mathcal{L}[i,j,l] = \frac{r-2}{r'-2}\mathcal{Q}[i,j,l] + \mathcal{B}[i,l] + B[j,l] \ , \qquad (4)$$

We observe that $\mathcal{L}[i,j,0] = Q_{i,j}$ and that $\mathcal{L}[i,j,l]$ is a lower bound on the Q matrix entries in rows $2^l i$ to $2^l(i+1)-1$ and columns $2^l j$ to $2^l(j+1)-1$:

$$\mathcal{L}[i,j,l] \quad \le \quad \min_{2^l i \le i' < 2^l(i+1),\ 2^l j \le j' < 2^l(j+1)} Q_{i'j'} \ .$$

The $\mathcal{L}$ values can be seen as a quad-tree, although it is implicitly defined by $\mathcal{Q}$ and $\mathcal{B}$.

2.2 Searching the Quad-Tree

We cannot simply search for the minimum valued leaf in $\mathcal{L}$ in the usual quad-tree search fashion, since we are no longer storing the minimum value in a range, but rather a lower bound on the minimum value. Instead, we will use the $\mathcal{L}[i,j,l]$ values to guide our search for the minimal Q_{ij} values.

Two approaches present themselves: A depth-first traversal of $\mathcal{L}$ with cut-offs when the lower bound is greater than a known Q_{ij} value, and priority queue based search that always expands the $\mathcal{L}[i,j,l]$ value with the lowest lower bound.

2.2.1 *Depth-First Search*

In the depth-first search approach, we simply explore $\mathcal{L}$ in a depth-first manner, looking for the minimal $\mathcal{L}[i,j,0]$ value. By definition, this is also the minimal Q_{ij} value. In itself, this will not speed up the search for the minimal value—although still in $O(r^2)$, traversing $\mathcal{L}$ is significantly slower than traversing the Q matrix to begin with—however, we can avoid exploring parts of the tree by cutting off searches of sub-trees. When we see a node $\mathcal{L}[i,j,l]$, whose lower bound is greater than an already seen Q value at the bottom level, we need not explore the sub-tree rooted in $\mathcal{L}[i,j,l]$ since none of the leaves in this tree will contain the minimal value.

The efficiency of this search greatly depends on how much of the tree can be discarded by cut-offs. In the worst case, no cut-offs are possible and we explore the entire $\mathcal{L}$, with a search time in $O(r^2)$, giving us a combined search time of $O(n^3)$. If, on average, we only need to explore $O(r)$ nodes, the combined search time is down to $O(n^2)$.

2.2.2 *Priority Queue Search*

In the priority queue approach, we use a priority queue to expand the $\mathcal{L}[i,j,l]$ nodes in a lowest-lower-bound-first order. This is based on the assumption that the lowest lower bound is more likely to contain the real lowest value. In each step, deletion of the minimum element in the priority queue gives us the unexplored node with the current lowerst lower bound, and each of the children of the node are then inserted into the priority queue. Once a deletion produces an element on level 0, we have found the minimal Q_{ij} value and need search no further.

As with the depth-first search, the efficiency of this search depends on how the lower bounds corresponds to the actual leaf-values in the tree. In the worst case, we need to explore the entire tree at a cost of $O(r^2 \log r)$, with a total search time of $O(n^3 \log n)$, while if, on average, we only search $O(r)$ nodes we have a cost of $O(r \log r)$, with a total search time of $O(n^2 \log n)$.

2.2.3 *Random Sampling*

Both the depth-first search and the priority queue search approaches can be extended with an initial random sampling, e.g., $O(r)$ entries of the Q_{ij} matrix, i.e. entries $\mathcal{L}[i,j,0]$ for random i and j. The minimum of these values can then be used as the initial cut-off value. For the depth-first search approach this allows the algorithm to make more qualified cut-offs already from the beginning of the search, whereas for the priority queue search approach the gain is minimal since the cut-off only reduces the number of insertions into the priority queue—the number of deletions remains unchanged.

2.3 Updating the Quad-Tree

In each join of two nodes, we need to delete two rows and two columns from Q—the two nodes we now join—and add one new row and column—for the new node. This update must also be represented in $\mathcal{L}$, which means that we need to update $\mathcal{Q}$ and $\mathcal{B}$. If we store the new row/column in one of the deleted rows/columns, say i, we need to update two rows/columns in Q' and all values in δ as follows (where $\overline{x}$

denotes the updated value of x):

$$\overline{d_{ik}} = (d_{ik} + d_{jk} - d_{ij})/2$$
$$\overline{\delta_k} = \delta_k + \overline{d_{ik}} - d_{ik} - d_{jk}$$
$$\overline{Q'_{ik}} = (r' - 2)\overline{d_{ik}} - (R'_i + R'_k)$$
$$\overline{Q'_{jk}} = \infty \qquad \text{(effectively deleting } j)$$

for all $k \neq i, j$.

Updating δ and Q' this way takes time $O(r)$. Rebuilding $\mathcal{B}$ from the new δ and updating $\mathcal{Q}$ based on the change of two rows/columns in Q' can also be done in time $O(r)$. Over the n iterations of the algorithm, this updating contributes $O(n^2)$ to the running time.

As the distance between r and r' grows, the information stored in Q' and R', and thus in $\mathcal{Q}$, diverges from the real values from Q and R. Consequently, the lower bounds in $\mathcal{L}$ becomes less accurate, and we expect to search more of $\mathcal{L}$ before we find a minimal leaf. It is therefore necessary to regularly update $\mathcal{L}$, by setting Q' to the current Q, updating R' and δ correspondingly, and rebuild $\mathcal{Q}$.

A rebuild takes time $O(r^2)$, so if we rebuild too frequently, there will be no gain in running time—rebuilding in each iteration, for instance, will result in an $O(n^3)$ algorithm. On the other hand, if we rebuild too infrequently, the search time will suffer due to the worse lower bounds.

We chose to rebuild each time we have processed a fraction of the remaining nodes, i.e. after $\frac{r'}{m}$ iterations, for some fixed m. Since the size of the matrices constructed decreases exponentially, this implies that we spend $O(n^2)$ time on rebuilding all in all. Together with the updating performed in each iteration, this gives a total update time of $O(n^2)$.

2.4 Limitations of approach

Essential for the heurestics presented in this section to be useful is that the derived lower bounds $\mathcal{L}[i, j, l]$ for a node in the quad-tree is close to the minimum value among the leafs in the subtree spanned by the quatree node. Comparing (3) and (4) this might be infeasible if the correction terms

$$\frac{r - r'}{r' - 2}(R'_i + R'_j) - (\delta_i + \delta_j)$$

span quite different values, since we use the minimum over all the correction values in the subtree. This is unfortunately the case what we expect for the R_i values. In the experiments in Section 4, we in Figure 3 observe that the performance of the above developed techniques, except for the depth first search without sampling, is essentailly the same as those obtained by the QuickTree algorithm. For details see Section 4).

The approach described in Section 3 addresses the above limitation by trying to reduce the distribution of the correction terms by adopting linear functions in r at the leaves of the quad-tree used.

3. APPROXIMATION OF Q USING LINEAR FUNCTIONS

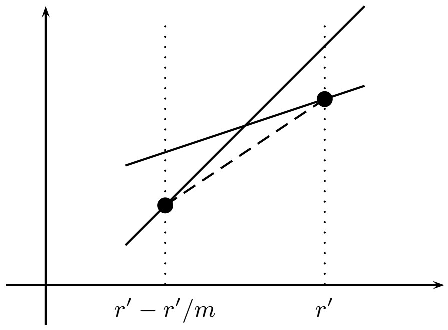


Figure 2: The lower bound linear function.

In (3) we based our search on an old Q matrix. In this section be base the approach on the following rewriting of (1) that only depends on old row sums R'_k:

$$\begin{aligned} Q_{ij} &= \left(d_{ij} - \frac{R'_i + R'_j}{r'}\right) r - 2d_{ij} + \frac{rR'_i}{r'} - R_i + \frac{rR'_j}{r'} - R_j \\ &= f_{ij}(r) + c_i(r) + c_j(r) \;, \end{aligned}$$

where

$$f_{ij}(r) = r\left(d_{ij} - \frac{R'_i + R'_j}{r'}\right) - 2d_{ij} \qquad (5)$$

$$c_i(r) = r\frac{R'_i}{r'} - R_i \;. \qquad (6)$$

The rewriting expresses Q_{ij} as a linear function f_{ij} over r plus some correction terms c_i and c_j—which, assuming $\frac{R_k}{r} \approx \frac{R'_k}{r'}$ for $k = i, j$, is likely to be small.

Note that f_{ij} only depends on the current value of d_{ij} and the values of r' and R'; we only need to update f_{ij} when i or j is joined in a new node, i.e., we only need to update a linear number of functions for each join.

We will below define a quad-tree with the f_{ij} functions at the leaves and where each internal node ideally should store the function that is the minimum over all the linear functions stored at the leaves of the subtree rooted at the node. Unfortunately this is not a linear function but a convex function consisting of piecewise linear functions. To achieve an efficient algorithm we instead for the interval of r values of interest maintain a lower bound for the convex function that is a linear function. .

Assume we decide to rebuild the structure after (at most) $\frac{r'}{m}$ iterations, for some fixed m. For two linear functions f_{ij} and $f_{i'j'}$, define $\min_m\{f_{ij}, f_{i'j'}\}$ to be the linear function that passes through the two points

$$\left(r' - r'/m, \min\left\{f_{ij}\left(r' - r'/m\right), f_{i'j'}\left(r' - r'/m\right)\right\}\right)$$

and

$$\left(r', \min\left\{f_{ij}\left(r'\right), f_{i'j'}\left(r'\right)\right\}\right) ,$$

as illustrated by Figure 2. Defined this way, $\min_m\{f_{ij}, f_{i'j'}\}$ is a lower bound for both of the functions until the next rebuilding:

$$\min_m\{f_{ij}, f_{i'j'}\}(r) \leq \min\{f_{ij}(r), f_{i'j'}(r)\}, \qquad (7)$$

for all $r \in [r' - r'/m, r']$. This minimum-operation is easily generalized to take the minimum of four functions, and we define a quad-tree $\mathcal{F}$ over the functions by:

$$\mathcal{F}[i,j,0] = f_{ij}(r)$$

$$\mathcal{F}[i,j,l] = \min_m \begin{cases} \mathcal{F}[2i,2j,l-1] \\ \mathcal{F}[2i+1,2j,l-1] \\ \mathcal{F}[2i,2j+1,l-1] \\ \mathcal{F}[2i+1,2j+1,l-1] \end{cases} \quad \text{for } l > 0$$

By induction on the number of minimum operations and a generalization of (7) we get

$$\mathcal{F}[i,j,l](r) \le \min_{2^l i \le i' < 2^l(i+1),\ 2^l j \le j' < 2^l(j+1)} f_{i'j'}(r),$$

for all $r \in [r' - r'/m, r']$.

We can use this tree, together with a binary *correction tree* $\mathcal{C}$ defined by

$$\mathcal{C}[k,0] = c_k(r)$$
$$\mathcal{C}[k,l] = \min\{\mathcal{C}[2k,l-1], \mathcal{C}[2k+1,l-1]\} \quad \text{for } l > 0$$

to define the implicit quad-tree

$$\mathcal{L}[i,j,l](r) = \mathcal{F}[i,j,l](r) + \mathcal{C}[i,l] + \mathcal{C}[j,l]$$

satisfying

$$\mathcal{L}[i,j,l](r) \le \min_{2^l i \le i' < 2^l(i+1),\ 2^l j \le j' < 2^l(j+1)} Q_{ij}$$

for the current r, assuming

1. $\mathcal{F}$ is updated along with the functions f_{ij} whenever i or j is joined,
2. $r \in [r' - r'/m, r']$,
3. $\mathcal{C}$ is current.

Condition 1 is necessary since f_{ij} depends on d_{ij} which changes when i or j is joined, and condition 2 is necessary because of the way the minimum operation is defined. Condition 3 simply states that since $\mathcal{C}[k,0]$ depend on the current value of R_k, it must be updated whenever a join is performed.

3.1 Searching

Before each iteration $\mathcal{L}$ we must rebuild a current version of $\mathcal{C}$, which takes time $O(r)$. After this, we can search for the minimal Q_{ij} using the lower bounds in $\mathcal{L}$, as described in the previous section, using either a depth-first search with cut-offs or a priority queue. If the number of nodes visited during a search on average is linear, the total search time is $O(n^2)$ for the depth-first approach, or $O(n^2 \log n)$ for the priority queue approach.

3.2 Updating

For each join we must update two rows/columns in $\mathcal{F}$ taking time $O(r)$ for a total of $O(n^2)$. Furthermore, we must completely rebuild the function tree $\mathcal{F}$ whenever r reaches $r' - r'/m$. For fixed m, this has a total cost of $O(n^2)$.

4. EXPERIMENTS

To evaluate the presented heuristics, we have implemented them in a tool, QuickJoin [7]. For evaluating the performance of QuickJoin we have compared the QuickJoin tree creation with the traditional neighbour-joining tree creation method, as implemented in the tool QuickTree [6].

The QuickJoin program takes a distance matrix of the taxa for input, and produces a tree as output. The QuickTree tool, likewise, can take a distance matrix as input and produce a tree as output. Additionally, it can take a multiple alignment as input, and produce either a distance matrix or a tree as output. When comparing the running time of the two tools, we call both tools with a distance matrix as input.

The data used for the evaluation were protein sequence alignments taken from the Pfam database [2, 1], and translated into distance matrices using QuickTree.

The platform where the experiments were conducted was a Linux RedHat 8.0 kernel 2.4.18-19.7, Pentium 4 2.66 GHz, 512 KB cache, 1 GB ram, both the QuickJoin program and the QuickTree program was compiled using gcc/g++ 3.1.1 with optimization -O3. The priority queues and stacks used were adopted from STL, and the quad-trees and binary trees were implemented as given by the equations.

To measure the running time of the programs we used the GNU time tool, the time report is the user time obtained by the `time -f %e` option (walltime in seconds).

For QuickJoin we examine both the heuristic based on a depth-first search with cutoffs and the heuristic based on a priority queue search. For QuickTree there is only one way of building trees (ignoring bootstrapping which is not currently supported in QuickJoin and therefore not included in our evaluation.)

4.1 Experimental Results

Figure 3 shows a plot of the walltime performance of the heuristics presented in Section 2 compared with quick-tree on the alignments from Pfam with 200-1000 sequences. It can be observed that the performance of the depth-first search heuristic without sampling has a quite unstable performance, whereas the other heurestics achieve a performance comparable with that of the QuickTree implementation.

Our main experimental results are shown in Figure 4, that shows the performance of of QuickTree, and QuickJoin with the depth-first search heuristic and with the priority queue heuristic with and without sampling as described in Section 3. The input for the runs is distance matrices for the Pfam alignments with 200 to 8000 sequences. It can be observed that the running time of all the presented heuristics are at the same level, and that all the heuristics output perform the QuickTree implementation.

Since the matrices used by the heuristics are symmetric, the symmetric part is not stored by QuickJoin. The way QuickJoin is implemented, the memory usage for representing the quad-tree is increased (by a factor of four) each time the number of taxa is increased to the next power of two. That is, the memory usage is close to constant between powers of twos, and grows by a factor of four when the input size crosses a power of two. As the memory usage grows, the number of pagefaults when running the program grows. This slows down the program, and is the explanation behind the increase in running time at $2^{12} = 4096$ in Fig. 4. A similar increase in running time is observed at $2^{11} = 2048$ when running QuickJoin on a machine with less ram. At $2^{13} = 8192$ we did not have enough ram to allocate sufficient memory for representing the quad-tree, and we therefore do have plots for alignments with more than 2^{13} sequences.

We have also used quickjoin on two datasets supplied by Georg Fuellen, Integrated Functional Genomics, University Hospital Muenster, who used neighbor-joining to produce large phylogenies as described in [4]. Dataset A is a multiple alignment of 1138 species, and dataset B is a multiple alignment of 1863 species. Both multiple alignments were been converted into corresponding distance matrices. Building trees using QuickTree took 8.29 sec for dataset A and 34.67 sec for dataset B. Building trees using QuickJoin took 3.09 (3.38) sec for dataset A and 6.50 (7.56) sec for dataset B when using the depth-first search (priority queue search) heuretstic.

Finally we have performed experiments on distance matrices where each entry is randomly distributed. The results for random entries are shown in Figure 5. It is observed that the performance of the presented techniques all are superior to the performance of QuickTree and that all the presented techniques except the depth-first search heuristic without sampling have comparable performance. These oberservations are consistent with results shown in Figure 4 for the Pfam data.

5. CONCLUSION

We have suggested heuristics for speeding up the search for $\min_{i,j} Q_{ij}$ in neighbour joining based on a quad-tree storing information about known lower bounds on parts of the Q matrix. All heuristics have a space bound of $O(n^2)$ and a time bound on the form

$$O(nS + U) ,$$

where S is the time used (on average) in each search and U is the time used for updating and rebuilding the quad-tree and other auxiliary data structures.

For the suggested heuristics, the update time has a worst case bound of $O(n^2)$ if we rebuild the quad-tree whenever we have halved the number of remaining nodes.

A worst case bound for S is $O(n^2)$, resulting in a combined $O(n^3)$ time bound for the heuristics, i.e., the same asymptotic bound as the original method.

We have conducted experiments, evaluating the performance of the heuristics implemented in QuickJoin on data from the Pfam database and have shown that the heuristics perform favourably compared to the traditional algorithm as implemented in QuickTree and achieves a significant speed up.

QuickTree is stated to be an optimized implementation of the Neighbor-Joining tree bulding algorithm [6]. We expect that if we apply a similar level of code optimization techniques to the implementation of QuickJoin used for the experiments we will be able achieve an improved performance increasing the gap between the performance of QuickTree and QuickJoin.

6. REFERENCES

[1] Pfam: Protein Families Database of Alignments and HMMs. `http://www.sanger.ac.uk/Software/Pfam/`.

[2] A. Bateman, E. Birney, L. Cerruti, R. Durbin, L. Etwiller, S. Eddy, S. Griffiths-Jones, K. Howe, M. Marshall, and E. Sonnhammer. The Pfam protein families database. *Nucleic Acids Res.*, 30:276–280, 2002.

[3] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval by composite key. *Acta Informatica*, 4(1):1–9, 1974.

[4] G. Fuellen, M. Spitzer, P. Cullen, and S. Lorkowski. BLASTing Proteomes, Yielding Phylogenies. *Silico Biology*, 3, 2003. To appear.

[5] O. Gascuel. A note on Sattath and Tversky's, Saitou and Nei's, and Studier and Keppler's algorithms for inferring phylogenies from evolutionary distances. *Mol. Biol. Evol.*, 11(6):961–963, 1994.

[6] K. Howe, A. Bateman, and R. Durbin. QuickTree: Building Huge Neighbour-Joining Trees of Protein Sequences. *Bioinformatics*, 18(11):1546–1547, 2002.

[7] T. Mailund. QuickJoin. `http://www.daimi.au.dk/~mailund/quick-join.html`.

[8] N. Nei and S. Kumar. *Molecular Evolution and Phylogenetics*, chapter 6.4, pages 103–110. Oxford University Press, 2000.

[9] N. Saitou. Reconstruction of Gene Trees from Sequence Data. *Methods in Enzymology*, 266:427–448, 1996.

[10] N. Saitou and M. Nei. The neighbor-joining method: A new method for reconstructing phylogenetic trees. *Mol. Biol. Evol.*, 4(4):406–425, 1987.

[11] K. St. John, T. Warnow, B. Moret, and L. Vawter. Performance study of phylogenetic methods: (unweighted) quartet methods and neighbor-joining. *J. Algorithms*, 48(1):173–193, 2003. (Special issue on best papers from SODA'01.).

[12] J. A. Studier and K. J. Keppler. A note on the neighbor-joining method of Saitou and Nei. *Mol. Biol. Evol.*, 5(6):729–731, 1988.

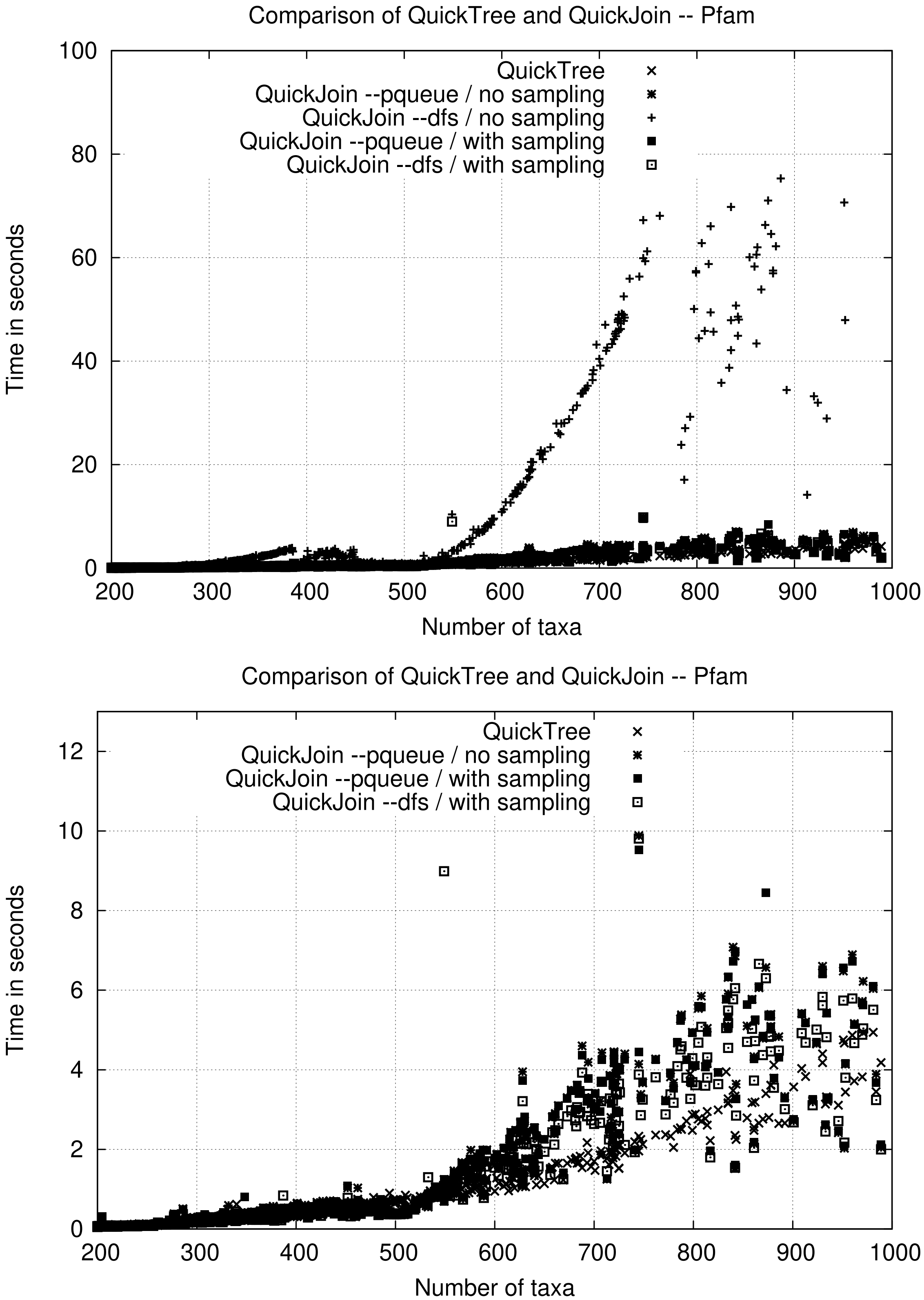


Figure 3: The plots shows the running time of QuickTree, and QuickJoin with the depth-first search heuristic and with the priority queue heuristic with and without sampling as described in Section 2. The input for the runs is distance matrices for the Pfam alignments with 200 to 1000 sequences.

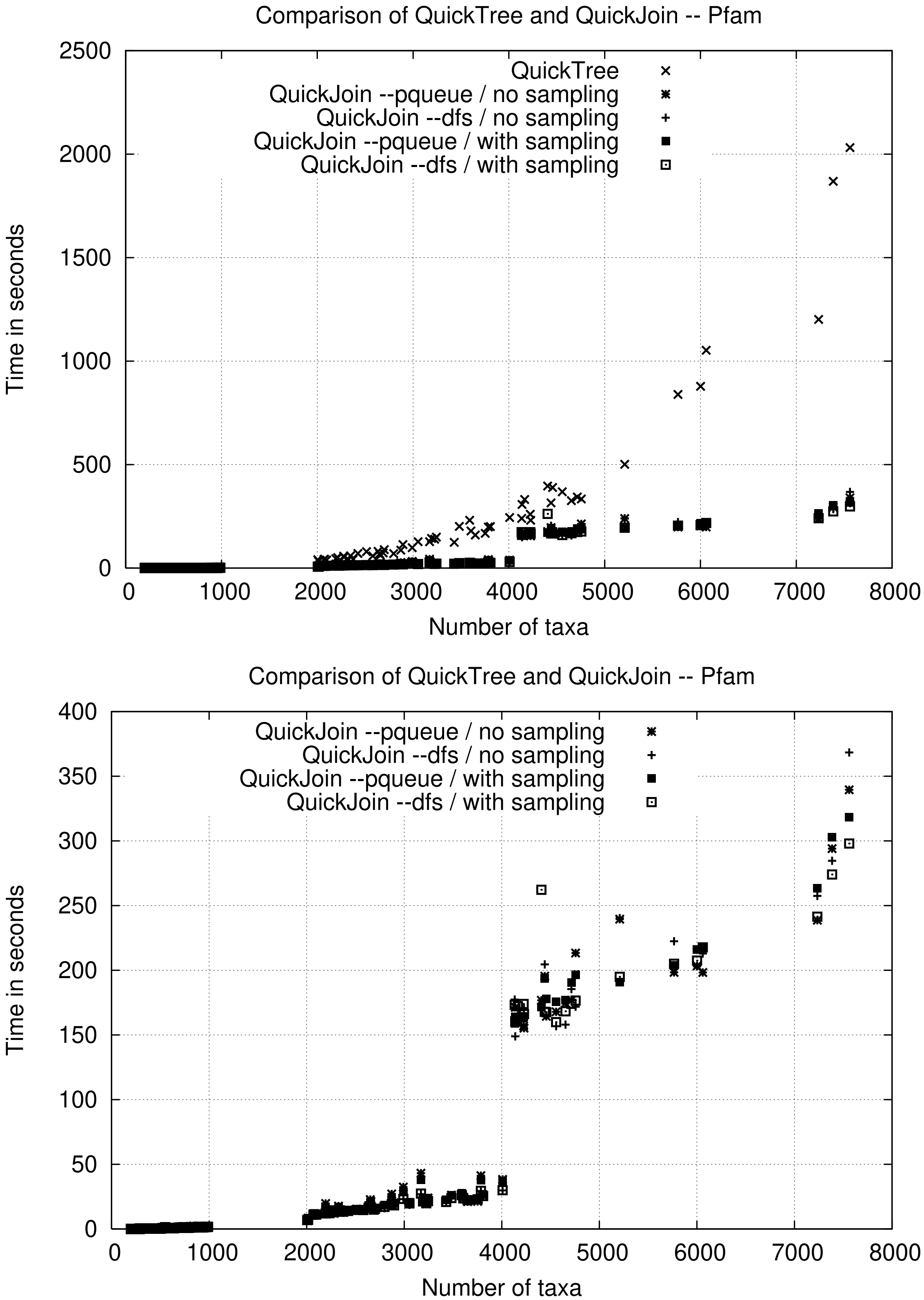


Figure 4: Running time of QuickTree, and QuickJoin with the depth-first search heuristic and with the priority queue heuristic with and without sampling as described in Section 3. The input for the runs is distance matrices for the Pfam alignments with 200 to 8000 sequences.

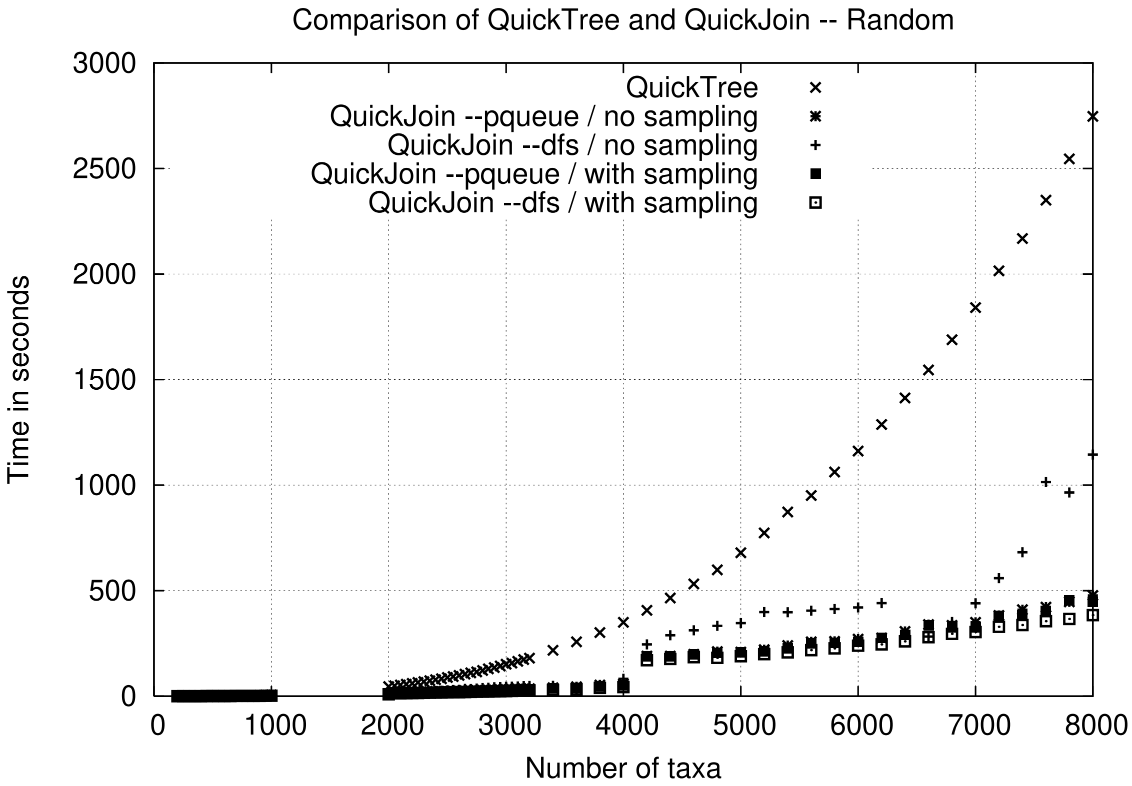


Figure 5: Running time of QuickTree, and QuickJoin with the depth-first search heuristic and with the priority queue heuristic with and without sampling as described in Section 3. The input are uniformly random distance matrices.