Journal of Bioinformatics and Computational Biology

Computing the All-Pairs Quartet Distance on a set of Evolutionary Trees

M. Stissing, T. Mailund, C. N. S. Pedersen and G. S. Brodal
Bioinformatics Research Center and Department of Computer Science, University of Aarhus, Denmark

R. Fagerberg
Department of Mathematics and Computer Science, University of Southern Denmark, Denmark

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Communicated by (xxxxxxxxxx)

We present two algorithms for calculating the quartet distance between all pairs of trees in a set of binary evolutionary trees on a common set of species. The algorithms exploit common substructure among the trees to speed up the pairwise distance calculations thus performing significantly better on large sets of trees compared to performing distinct pairwise distance calculations, as we illustrate experimentally, where we see a speedup factor of around 130 in the best case.

Keywords: Evlutionary trees; quartet distance

1. Introduction

In biology, trees are widely used for describing the evolutionary history of a set of taxa, e.g. a set of species or a set of genes. Inferring an estimate of the (true) evolutionary tree from available information about the taxa is an active field of research and many reconstruction methods have been developed, see e.g. Felsenstein [1] for an overview. Different methods often yield different inferred trees for the same set of species, as does using different information about the species, e.g. different genes, when using the same method. To reason about the differences between estimates in a systematic manner several distance measures between evolutionary trees have been proposed. [2–6] Given a set of estimates of evolutionary trees on the same set of species, the differences between the trees can then be quantified by all the pairwise distances under an appropriate distance measure.

This paper concentrates on computing the distance between all pairs of trees in a set of unrooted fully resolved (i.e. binary) evolutionary trees on a common set of species using a distance measure called the quartet distance. [3] For an evolutionary

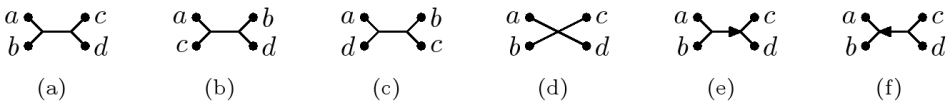


Fig. 1. Figures (a)–(d) show the four possible quartet topologies of species a, b, c, and d. Topologies $ab|cd$, $ac|bd$, and $ad|bc$ are *butterfly* quartets, while topology $\frac{a}{b} \times \frac{c}{d}$, is a *star* quartet. For binary trees, only the butterfly quartets are possible. Figures (e) and (f) show the two ordered butterfly quartet topologies induced by the butterfly quartet topology in (a).

tree, the *quartet topology* of four species is determined by the minimal topological subtree containing the four species. The four possible quartet topologies of four species are shown in Fig. 1 (a)–(d). In a fully resolved tree only the three fully quartet topologies can occur (Fig. 1 (a)–(c)). Given two evolutionary trees on the same set of n species, the *quartet distance* between them is the number of sets of four species for which the quartet topologies differ in the two trees. For binary trees, the quartet distance can be calculated in time $O(n \log n)$, where n is the number of species.[7] A simpler algorithm with running time $O(n \log^2 n)$ has been implemented in the tool *QDist*.[8] For arbitrary degree trees, the quartet distance can be calculated in time $O(n^3)$ or $O(n^2 d^2)$, where d is the maximum degree of any node in any of the two trees.[9]

As fully resolved trees are the focus of most reconstruction algorithms for evolutionary trees, we in this paper develop an algorithm for computing the pairwise distances between k fully resolved trees on a common set of species. Our algorithm exploits similarity between the input trees to improve on the straightforward approach of performing the k^2 pairwise comparisons independently which has $O(k^2 t)$ running time, where t is the time for comparing two trees ($O(n \log n)$ if the algorithm from Brodal et al.[7] is used). The worst case running time of our approach remains $O(k^2 t)$, if the input trees share very little structure, but experiments show that our algorithms achieve significant speedups of more than a factor 100 on more related trees. Our method has been implemented as an extension to the existing tool QDist by Mailund and Pedersen[8].

Comparing multiple trees can be used to study the tree space as in Hillis et al.[10]. Comparing multiple trees can also be applied when constructing supertrees, i.e. constructing a larger tree from a set of smaller source trees. E.g. the QFIT method of the Clann program[11] essentially evaluates the quality of a predicted supertree by comparing it against every source tree in terms of the quartet distance.

2. The Pair-wise Quartet Distance

The $O(n \log^2 n)$ time algorithm developed by Brodal et al.[12] serves as a basis for our all-pairs algorithm. The $O(n \log n)$ time algorithm works just as well, but as there is no known implementation of it, we settled for the simpler $O(n \log^2 n)$ time algorithm. In this section we describe the algorithm for computing the quartet

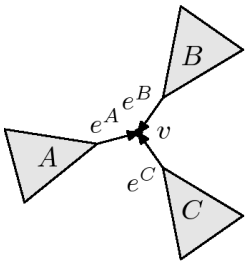


Fig. 2. The inner node v induces oriented quartets through each of the three incident edges.

distance between a pair of trees. Two basic ideas are used in the algorithm: instead of counting the number of quartet topologies that differs between the two trees, the algorithm counts the number of quartet topologies that are shared and subtracts this number from the total number possible of quartets, $\binom{n}{4}$, and instead of counting shared quartet topologies, it counts the shared *ordered* topologies. An (unordered) butterfly quartet topology, $ab|cd$ induces two ordered quartet topologies $ab \rightarrow cd$ and $ab \leftarrow cd$, by the orientation of the middle edge of the topology, as shown in Fig. 1, (e) and (f). Clearly there are twice as many ordered quartet topologies, and the quartet distance can thus be obtained by subtracting from $2\binom{n}{4}$ the number of shared ordered quartet topologies and then dividing by 2.

Given an ordered edge, e, let A denote the tree behind e, and B and C denote the two trees in front of e. Each such edge induces a number of oriented quartets: $aa' \rightarrow bc$ where $a, a' \in A$, $b \in B$, and $c \in C$, and each oriented quartet is induced by exactly one oriented edge in this way. The number of oriented quartets induced by e is $\binom{|A|}{2} \cdot |B| \cdot |C|$ where $|X|$ is used to denote the number of leaves in the tree X.

For each inner node, v, each of the incidence oriented edges, see Fig. 2, induces oriented quartets, and the number of oriented quartets induced by the node is

$$\binom{|A|}{2} \cdot |B| \cdot |C| + \binom{|B|}{2} \cdot |A| \cdot |C| + \binom{|C|}{2} \cdot |A| \cdot |B| \,. \tag{1}$$

When comparing two trees, T_1 and T_2, we can consider a pair of inner nodes, v_1 in T_1 and v_2 in T_2, with sub-trees A_1, B_1, C_1 and A_2, B_2, C_2 and incident edges e_1^A, e_1^B, e_1^C and e_2^A, e_2^B, e_2^C, respectively, as in Fig. 2. The shared oriented quartets induced by edges e_1^A and e_2^A are the oriented quartets $aa' \rightarrow xy$ where $a, a' \in A_1 \cap A_2$ and x and y are found in different subtrees in front of e_1^A and e_2^A, respectively. The number of shared oriented quartets induced by e_1^A and e_2^A, $\mathrm{count}(e_1^A, e_2^A)$ is then:

$$\binom{|A_1 \cap A_2|}{2} \cdot \Big(|B_1 \cap B_2| \cdot |C_1 \cap C_2| + |B_1 \cap C_2| \cdot |C_1 \cap B_2| \Big) \,, \tag{2}$$

where $|X_1 \cap Y_2|$, with a slight abuse of notation, denotes the number of shared leaves in sub-trees X_1 and Y_2. The number of shared oriented quartets for nodes v_1 and v_2

Fig. 3. The coloured leaves for an inner node, v_1 in T_1 and the same colouring for inner node v_2, in T_2.

is the sum of the shared quartets of their incident edges:

$$\text{count}(v_1, v_2) = \sum_{X \in \{A,B,C\}} \sum_{Y \in \{A,B,C\}} \text{count}(e_1^X, e_2^Y) \tag{3}$$

and the total count of shared quartets is the sum of counts for all pairs of nodes v_1 and v_2.

We can express (2) slightly differently: Let node v_1 be an inner node in T_1 with associated sub-trees A, B and C as in Fig. 2. We can then colour all leaves in A with the colour $\mathcal{A}$, the leaves in B with the colour $\mathcal{B}$, and the leaves in C with the colour $\mathcal{C}$. The leaves in T_2 can be coloured with the same colours as in T_1, and for any inner node, v_2 in T_2, the sub-trees will then contain a number of leaves of each colour, see Fig. 3. The size of the intersections of leaves in the sub-trees in T_1 and T_2, respectively, is now captured by the colouring. The oriented quartets induced by both e^A and e^1 (see Fig. 3) are the quartets $aa' \to bc$ where a and a' are coloured $\mathcal{A}$ and present in the first subtree of v_2 and one of b, c is coloured $\mathcal{B}$ and the other $\mathcal{C}$ and present in the two other subtrees respectively. Let $\mathbf{a}(i), \mathbf{b}(i), \mathbf{c}(i)$, $i = 1, 2, 3$ denote the number of leaves with colour $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ in the three sub-trees of v_2. The "colouring analogue" to (2) is then

$$\text{count}(e^A, e^1) = \binom{\mathbf{a}(1)}{2} \cdot \Big(\mathbf{b}(2) \cdot \mathbf{c}(3) + \mathbf{b}(3) \cdot \mathbf{c}(2)\Big) . \tag{4}$$

Similar expressions can be derived for the remaining pairs of edges.

A naïve approach simply colours the leaves with colours $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, for each node in T_1, and then, for each node in T_2, counts the numbers $\mathbf{a}(i), \mathbf{b}(i), \mathbf{c}(i)$ and evaluates the sum in (3). The time complexity of this is the time it takes for the colouring of leaves according to the inner node in T_1 plus the time it takes to count $\mathbf{a}(i), \mathbf{b}(i), \mathbf{c}(i)$ and evaluate the sums in T_2. Since the colouring is a simple matter of traversing T_1, and assuming we connect the leaves of T_1 with the corresponding leaves in T_2 with pointers, the colouring takes time $O(n)$ for each node of T_1 giving a total colouring time of $O(n^2)$. If we root T_2 in an arbitrary leaf, we can count $\mathbf{a}(i), \mathbf{b}(i), \mathbf{c}(i)$ in a depth-first traversal: we annotate each edge with the number of leaves below it with colour $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, respectively, and for a node v_2, the counts

out of each child-edge is immediately available from this annotation. The counts on the edge to the root can be obtained from the sum of the counts on the children-edges if we remember the total number of leaves in each colour. This way we can count and then evaluate the sum for all nodes in time $O(n)$ for each colouring, with a total counting time of $O(n^2)$, giving all in all a running time of $O(n^2)$.

The algorithm in Brodal et al.[12] improves on this using two tricks: it reuses parts of the colouring when processing the nodes of T_1 and uses a "smaller part" trick to reduce the number of re-colourings, and it uses a hierarchical decomposition tree to update the counts in T_2 in a clever way thus reducing the counting time. As a preprocessing step, the first tree is rooted in an arbitrary leaf, r, and traversed depth first to calculate the size of each sub-tree. This size is stored in the root of each sub-tree, such that any inner node, v, in constant time can know which of its children is the root of the smaller and the larger sub-tree (with an arbitrary but fixed resolution in case of ties). When counting, initially, r is coloured $\mathcal{C}$ and all other leaves are coloured $\mathcal{A}$. The tree is then traversed, starting from r, with the following invariant: before processing node v, all leaves in the subtree rooted in v are coloured $\mathcal{A}$ and all remaining leaves are coloured $\mathcal{C}$; after the processing, all leaves in v's subtree are coloured $\mathcal{C}$. The processing of node v consists of first colouring the smaller subtree of v with the colour $\mathcal{B}$ and then counting the number of shared quartets induced by v. Then the smaller subtree is recoloured $\mathcal{C}$ allowing a calculation of the quartets shared in the larger subtree—since all the leaves in the larger subtree have colour $\mathcal{A}$ and the rest of the leaves now have the colour $\mathcal{C}$. After this, the leaves in the larger subtree have, according to the invariant, been coloured $\mathcal{C}$. The leaves in the smaller subtree are then recoloured $\mathcal{A}$ thus enabling a recursive count of the shared quartets in the smaller sub-tree. The number of shared quartets in v's subtree is the sum of the three counts. For the time usage on colouring, observe that each leaf is only coloured initially and then only when it is part of a smaller tree. Since each leaf can at most be part of $O(\log n)$ smaller trees, the total colouring time is in $O(n \log n)$.

The hierarchical decomposition trick is a balanced tree-structure that enables constant time counting, but has an updating cost associated with re-colourings: updating the colour of a leaf takes $O(\log n)$ time. Ultimately this yields the $O(n \log^2 n)$ running time of the algorithm. The details of this tree-structure is not important for the algorithm we develop in this paper—it can be reused in this algorithm in a straight forward manner—so we will not go into details about them here and instead refer to Brodal et al.[12]

3. The All Pairs Quartet Distance

We consider four algorithms for calculating the all-pairs quartet distance between k trees. The two simplest of these simply perform $O(k^2)$ single comparisons between two trees using time $O(n^2)$ and $O(n \log^2 n)$ per comparison, respectively. These are the algorithms we will attempt to improve upon by exploiting similarity between

the compared trees to obtain a heuristic speedup. The use of similarity utilizes a *directed acyclic graph* (DAG) representation of the set of trees. Given our set of k trees, we can root all trees in the same arbitrary leaf, i.e. a leaf label is chosen arbitrarily and all trees are rooted in the leaf with that label. Given this rooting, we can define a representation of the trees as a DAG, D, satisfying:

(1) D has k nodes with in-degree 0: $r_1, r_2, \ldots, r_k$, one for each root in the k trees; we will refer to these as the *roots*.
(2) D has $n-1$ nodes with out-degree 0, one for each of the non-root leaves in the trees; these DAG leaves are labelled with the $n-1$ non-root leaf labels; these are the *leaves*.
(3) For each tree T there is an embedding of T in D, in the sense that there is a mapping $e_i : v(T) \to v(D)$ such that the root of T is mapped to the root in D representing T, r_i, the leaves of T are mapped to the leaves of D with the same labels, and for each edge in T, (v, v'), the image of e_i, $(e_i(v), e_i(v'))$, is an edge in D.
(4) For any non-root nodes v_1 and v_2 from trees T_1 and T_2, respectively, if the tree rooted in v_1 is isomorphic to the tree rooted in v_2, then v_1 and v_2 are represented by the same node in D, $e_1(v_1) = e_2(v_2)$.

Conditions 1–3 ensures that each of the k trees are represented in D, in the sense that tree T_i can be extracted by following the directed edges from r_i to the leaves. Condition 4 ensures that this is the minimal such DAG. Condition 1 is implied by the remaining if no two trees in the set are isomorphic. Since isomorphic trees will have distance 0, and this isomorphism is easy to check for when building the DAG as described below, we assume that no trees are isomorphic and thus that merging the trees will automatically ensure Condition 1.

We can build the DAG iteratively by merging one tree at a time into D. Initially, we set $D := T_1$, then from $i = 2$ to $i = k$ we merge T_i into D using a depth first traversal of T_i, for each node inserting a corresponding node in the DAG if it isn't already present. Since the set of leaves is the same for all trees, each leaf node can be considered already merged into D. For each inner node v, with children u and w, recursively merge the subtrees of u and w, obtaining the mapped DAG nodes $e(u)$ and $e(w)$. If these share a parent in D, v is mapped to that parent, otherwise a new node $e(v)$ is inserted in the DAG. The test for a shared parent of two nodes can be made efficiently by keeping a table mapping pairs of DAG nodes to their shared ancestor—two DAG nodes can never share more than one parent, since such two parents would be merged together. This table can be kept updated by inserting $\big(e(u), e(w)\big) \mapsto e(v)$ whenever we insert $e(v)$ as above. Note that each inner DAG node has out-degree 2.

We observe that if two tree nodes map to the same DAG node, they induce the same oriented quartets: if $e_i(v_i) = e_j(v_j)$ for inner nodes v_i in T_i and v_j in T_j, then, by Condition 3, the subtrees of v_i and v_j are isomorphic, i.e. one child of v_i is isomorphic with one of v_j and the other child with the other of v_j. Thus, in a

colouring as on the left in Fig. 3, if the isomorphic child-trees are coloured the same colour, the result is the same leaf-colourings in the two trees T_i and T_j. By counting the oriented quartets in the DAG, we reduce the amount of work necessary, since merged nodes (tree nodes mapped to the same DAG node) need only be processed once, not once for each tree.

Calculating the pairwise quartet distance between a set of trees $S = \{T_1, T_2, \ldots, T_k\}$ and a single tree T can be done by first merging the trees in S into a DAG D. We then preprocess D depth-first for each of its k roots calculating sizes of subtrees, such that each inner node v in D will know which of its two children is root in the smaller and the larger subtree. This enables us to use the same "smaller part" trick when colouring the subtrees of D as is used when comparing two trees. The algorithm starts by colouring all roots of D using $\mathcal{C}$. Now, for each root r_i in D, the algorithm colours all leafs in the embedded tree T_i rooted at r_i, using the colour $\mathcal{A}$, and traverses T_i recursively, following the colouring schema and invariants used in the one-against-one tree comparison. However, whenever we process a node in the DAG, we store the sum of the count for that node plus the count for all nodes below it (this is done recursively). Since all counts for that node will be the same, regardless of which root we are currently processing, we can reuse the count if we see the node again when processing later trees. The counting as such can run exactly as described in the previous section, except that when a previously counted node v is met, we simply reuse the count and colour the subtree rooted at v with $\mathcal{C}$ in accordance with the stated invariant. It is still possible to use a hierarchical decomposition of the tree T to speed up the counting. Since we only recursively count each node in D once, not each time a node appears in one of the trees in S, we potentially save a significant amount of time if the trees are closely related and thus D is comparably small.

When calculating the all-pairs distance between k trees, this approach will still have an $O(k^2 t)$ worst case time bound, where t is the time for comparing two trees. In practice though, especially for highly similar trees, we suspect that this algorithm will outperform the naïve approach. In the case of similar trees, that is comparing $S = \{T_1, T_1, \ldots, T_1\}$ of k identical trees and a tree T the time used is $O(kn)$ for building D, $O(t)$ for comparing the first embedded tree of D with T and $O(kn)$ for comparing and recolouring the rest of the trees embedded in D with T (this covers the cost of constructing and updating a hierarchical decomposition tree as well). As we do this k times in order to get the all-pairs distance we obtain a total running time of $O(k(kn+t))$ which is superior to the naïve approach when n is a true lower bound for t, i.e. $t \in \omega(n)$. We expect that nontrivial comparisons will be somewhere in between the stated bounds.

Calculating the distance between all pairs of trees, or more generally the pairwise distances between two sets of trees, $S = \{T_1, T_2, \ldots, T_k\}$ and $S' = \{T'_1, T'_2, \ldots, T'_{k'}\}$ might clearly be done as stated above: for each $T_i \in S$ calculate the k' distances between T_i and S' using the algorithm described above. Merging both sets into DAGs, however, will let us exploit similar tree structure in both sets.

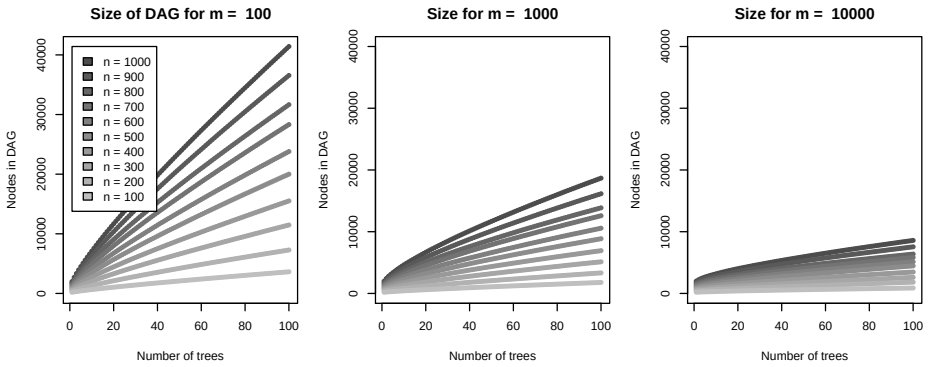


Fig. 4. Sizes of DAGs constructed. As predicted, as m grows, the size of the DAG is reduced, and as shown the number of nodes in the DAG is considerably lower than the number of nodes in the sets of trees. (By construction, leaves in a DAG are shared, meaning that the number of nodes in a DAG will be approximately less than half the number of nodes in the corresponding trees.) Note that for $k = 100$, $n = 1000$ and $m = 10000$, the collection of trees contain $100 \cdot 1998$ nodes, whereas the average size of the DAG is 8588. This is $\frac{8588}{100 \cdot 1998} \approx \frac{1}{23}$ the size of the corresponding set of trees or just $\frac{8588}{1998} \approx 4.3$ times the size of one single tree.

Let D and D' be the DAG representations of S and S', respectively. Finding the distances between all pairs of trees can be done by counting, for each tree T_i embedded in D, the number of shared quartets between T_i and each tree in S'. This amounts to running through D, colouring recursively as stated above, and counting how many quartets are shared between $v \in v(D)$ and each T' in S'. Storing this set of counts in v lets us process each v only once. Given a colouring, calculating how many quartets, for each T' embedded in D', is compatible with this colouring, may reuse the technique from above. Running through D' recursively, root for root, storing the number of quartets compatible with the current colouring in the subtree rooted at v' for each v' in D', enables reusing this number when v' is again processed. The worst case time for this approach remains $O(k^2n^2)$. If we consider identical trees, the time used in calculating the all-pairs quartet distance between these is $O(n^2+k)$ for comparing the first tree with all others and $O(k(k+n))$ for comparing and colouring the remaining pairs. In total, this is $O(n^2+k^2)$. In real life examples, we suspect a running time somewhere between these bounds.

4. Experimental Results

The new algorithms are expected to obtain significant speedups when comparing multiple trees of high similarity. In this section we report on experiments to validate this. An essential assumption for the expected speedup of using DAGs is that the trees share much structure, i.e. that the number of nodes in the DAGs are significantly smaller than the number of nodes in all pairs of trees. To evaluate this, we calculated the sizes of the DAGs obtained by merging the trees, see

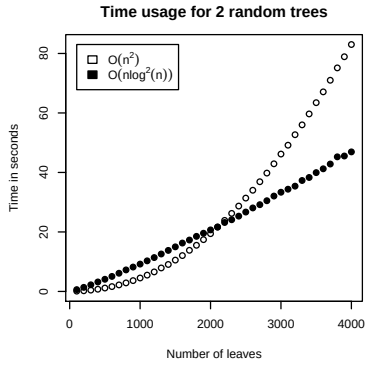


Fig. 5. Time used in calculating the quartet distance between two trees. Random trees were used as none of the algorithms should be favoured. The graphs show the average time of 30 runs for each size.

Fig. 4. The results of these experiments are very encouraging. The trees were obtained using the following method: First, to construct similar trees of size n (trees with n leaves), a phylogenetic tree of size n is constructed (using the tool *r8s* [13]). This tree is then used in simulating evolution along its phylogeny, hence evolving n sequences of a given length m (using the tool *Seq-Gen* [14]). The distances between these sequences give rise to an $n \times n$ matrix of distances (calculated using *ednadist*). When neighbour-joining (as implemented in *QuickJoin* [15]) is applied to this matrix it will result in a tree of size n. This process is repeated (using the same phylogenetic tree) k times to obtain k trees. The length of the simulated sequences, m, influences the similarity of the trees: as m grows, the sequences evolved will contain more of the information contained in the original phylogenetic tree. Ultimately the k trees developed will be nearer the original phylogenetic tree and thus the pairwise similarity of the k trees constructed will grow as m grows. For each $n \in \{100, 200, \ldots, 1000\}$, $k = 100$ trees of sequence lengths $m \in \{100, 1000, 10000\}$ were constructed. Construction of data was repeated 30 times. Experiments were run on these 30 sets of data except where otherwise stated.[a]

To have a baseline to compare our new algorithms with, we then timed the two tree-against-tree algorithms. First, we compared the simple $O(n^2)$ time algorithm with the more complex $O(n \log^2 n)$ time algorithm, both as implemented in the *QDist* tool.[8] Since the $O(n \log^2 n)$ time complexity is achieved through a number of involved polynomials in the nodes of the hierarchical decomposition tree, we expect it to have a significant overhead compared to the much simpler $O(n^2)$ time algorithm, which is verified in Fig. 5. Observe that for $n = 100$ the simpler algorithm is 8 times faster than the more complex. For $n = 1000$ this factor has shrunk to 2 and for $n \geq 2000$ the complex algorithm is preferable.

[a]Six identical, Intel Pentium 4 1.80 GHz, 512 MB, machines were used for the experiments.

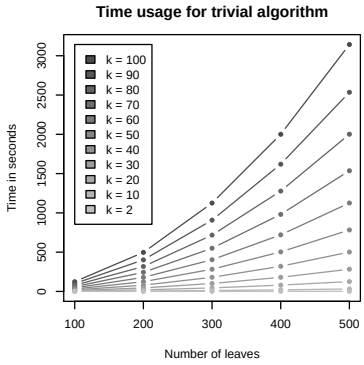
This is not the full story, however. The $O(n \log^2 n)$ time algorithm, as implemented in the tool *QDist* caches information about polynomial manipulation when handling the hierarchical decomposition tree[8]. This caching makes the *QDist* tool faster for more similar trees and the running time of *QDist* is therefore dependent on m. Our experiments shown in Fig. 6 indicate that the simpler, but $O(k^2 n^2)$ time, is faster than the more complex, but $O(k^2 \cdot n \log^2 n)$ time, algorithm for tree sizes up to at least 500 leaves.

Based on these experiments, we expect the DAG-against-DAG algorithm to outperform the DAG-against-tree algorithm for similar sizes of trees, and indeed, as shown in Fig. 7, this is the case. The DAG-against-tree algorithm (Fig. 7(a)) is clearly superior to the tree against hierarchical decomposition algorithm in all cases, and as m grows this superiority becomes more and more apparent, but compared with the DAG-against-DAG algorithm (Fig. 7(b)) it performs rather poorly (notice that the number of leaves for Fig. 7(b) grows from 100 to 1000 while for Fig. 7(a) it grows only to 500). The DAG-against-DAG algorithm, however, achieves a speedup factor around 130, compared to the simple (but faster) tree-against-tree algorithm for $k = 100$, $n = 500$ and $m = 10000$.

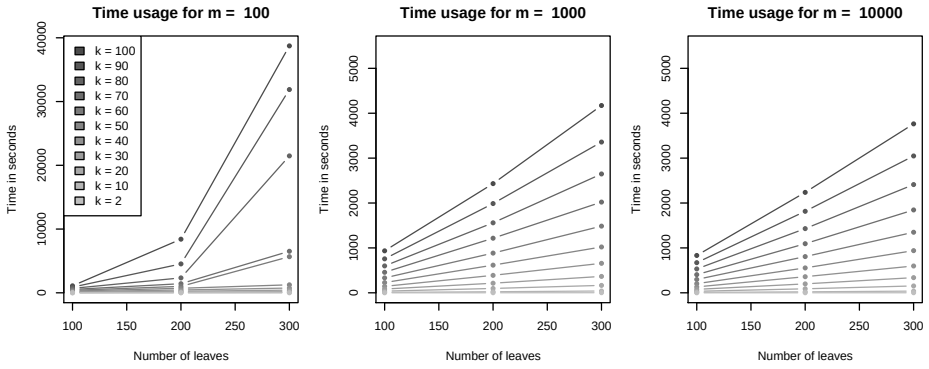
5. Conclusions

We have presented two new algorithms for computing the pairwise quartet distance between all pairs of a set of trees. The algorithms exploit shared tree structure on the tree set to speed up the computation by merging the set into a DAG and memorize counts in each node in the DAG. Our experiments show that the algorithms developed work well in practise (on simulated, but realistic trees), with a speedup factor of around 130 in the base case.

Calculating the quartet distances between trees not sharing the same set of leafs is not trivially done using the above construction. Two such trees might be compared by removing any leaf not in both trees prior to calculation or by using a fourth colour, $\mathcal{D}$, indicating not shared leaves. When comparing many such trees it is apparent that the DAG approach is not applicable since, assuming we adopt the same mapping of tree nodes to DAG nodes as used above, two tree nodes mapping to the same DAG node does not necessarily induce the same set of quartets and therefore it is not clear how to reuse the count of nodes in the DAG. This might be interesting looking further in to.

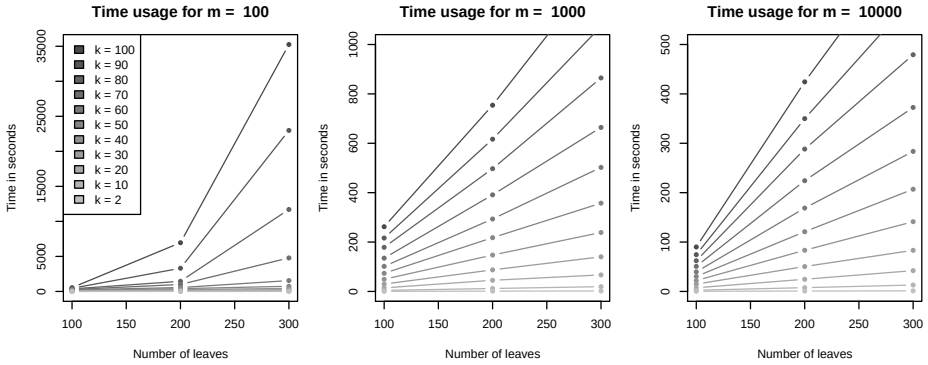


(a) $O(k^2n^2)$

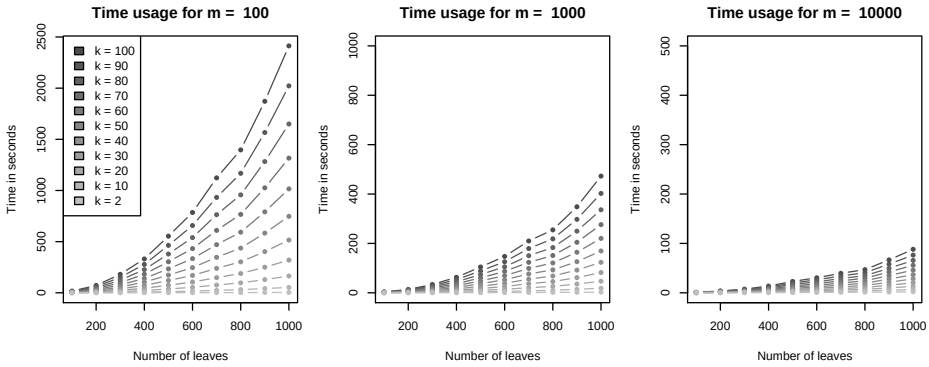


(b) $O(k^2 \cdot n \log^2 n)$

Fig. 6. The performance of the two $O(k^2t)$ time algorithms. (a) Time used in tree against tree comparison, i.e. applying the $O(n^2)$ time algorithm one against one for each pair of trees taken from the k trees. As this algorithm is independent of the similarity of the trees (the size of m) only $m = 100$ is shown. This experiment has only been repeated six times with $n \in \{100, 200, 300, 400, 500\}$ due to its quite time consuming nature and the above mentioned independence of similarity. For k and n the time used is seen to be close to $1.23 \cdot 10^{-6} \cdot k^2 \cdot n^2$ seconds. This method seems most appropriate for fewer smaller trees of little similarity due to its $O(k^2n^2)$ time complexity. (b) Time used in tree against hierarchical decomposition tree comparison. This experiment consists of constructing hierarchical decomposition trees for each of the k trees and running the original $O(n \log^2 n)$ time algorithm for each possible pair of tree and hierarchical decomposition tree. This experiment has only been repeated six times with values of $n \in \{100, 200, 300\}$, because of its time consumption. (We note that the graph considering $m = 100$ is not as regular as the others. This might be due to the relatively few experiments run.)



(a) DAG against hierarchical decomposition tree.



(b) DAG against DAG.

Fig. 7. The performance of the two DAG-based algorithms. (a) The DAG-against-tree algorithm. Here we construct a DAG D on top of the k trees as well as constructing a set S of hierarchical decomposition trees, each constructed from one of the k trees. Calculating the k^2 quartet distances is accomplished by comparing each $H \in S$ against D. Each experiment is only repeated six times due to the long running times. The graphs show that as m and k grows the algorithm "speeds up" and in the extreme case of $m = 10000$ and $k = 100$ the algorithm is approximately 50 times faster than the same algorithm run on $m = 100$ and $k = 100$. (b) The DAG-against-DAG algorithm. Here we construct a DAG D on top of the k trees and comparing D against a copy of itself. The experiments show that DAG against DAG is the fastest way of calculating the quartet distances (for the values of k, n and m examined) and that the speedup gained in comparison with the naïve tree against tree algorithm is in the vicinity of a factor of 130 for $k = 100$, $n = 500$ and $m = 10000$.

References

1. J. Felsenstein. *Inferring Phylogenies*. Sinauer Associates Inc., 2004.
2. B.L. Allen and M. Steel. Subtree transfer operations and their induced metrics on evolutionary trees. *Annals of Combinatorics*, 5:1–13, 2001.
3. G. Estabrook, F. McMorris, and C. Meacham. Comparison of undirected phylogenetic trees based on subtrees of four evolutionary units. *Syst. Zool.*, 34:193–200, 1985.
4. D. F. Robinson and L. R. Foulds. Comparison of weighted labelled trees. In *Combinatorial mathematics, VI*, Lecture Notes in Mathematics, pages 119–126. Springer, 1979.
5. D. F. Robinson and L. R. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53:131–147, 1981.
6. M. S. Waterman and T. F. Smith. On the similarity of dendrograms. *Journal of Theoretical Biology*, 73:789–800, 1978.
7. G.S. Brodal, R. Fagerberg, and C.N. S. Pedersen. Computing the quartet distance between evolutionary trees in time $O(n \log n)$. *Algorithmica*, 38:377–395, 2003.
8. T. Mailund and C.N.S. Pedersen. QDist—Quartet distance between evolutionary trees. *Bioinformatics*, 20(10):1636–1637, 2004.
9. C. Christiansen, T. Mailund, C.N.S. Pedersen, and M. Randers. Algorithms for computing the quartet distance between trees of arbitrary degree. In *Proc. of WABI*, volume 3692 of *LNBI*, pages 77–88. Springer-Verlag, 2005.
10. David M. Hillis, Tracy A. Heath, and Katherine St. John. Analysis and visualization of tree space. *Systematic Biology*, 54(3):471–482, 2005.
11. C. J. Creevey and J. O. McInerney. Clann: investigating phylogenetic information through supertree analyses. *Bioinformatics*, 21(3):390–392, Feb 2005. doi: 10.1093/bioinformatics/bti020.
12. G.S. Brodal, R. Fagerberg, and C.N.S. Pedersen. Computing the quartet distance between evolutionary trees. In *Proc. of ISAAC*, volume 2223 of *LNCS*, pages 731–742. Springer-Verlag, 2001.
13. M.J. Sanderson. r8s: inferring absolute rates of molecular evolution, divergence times in the absence of a molecular clock. *Bioinformatics*, 19(2):301–302, 2003.
14. A. Rambaut and N.C. Grassly. Seq-gen: an application for the monte carlo simulation of dna sequence evolution along phylogenetic trees. *Computer Applications in the Biosciences*, 13(3):235–238, 1997.
15. T. Mailund and C.N.S. Pedersen. QuickJoin—fast neighbour-joining tree reconstruction. *Bioinformatics*, 20(17):3261–3262, 2004. ISSN 1367-4803.

Martin Stissing received his Master's degree from the Department of Computer Science, University of Aarhus, Denmark, in 2006 and now works as a software developer at Mjølner Informatics A/S.

Thomas Mailund received his Master's degree in 2000 and his PhD in 2003, both from the Department of Computer Science, University of Aarhus, Denmark. From 2003 to 2006 he held a position as assistant professor at the Bioinformatics Research Center, University of Aarhus. From 2006 to 2007 he did a post doc. at the Department of Statistics, Oxford University, UK, and since 2007 he is again affiliated with the Bioinformatics Research Center, University of Aarhus.

Christian N. S. Pedersen received his Master's degree in 1997 and his PhD in 2000, both from the Department of Computer Science, University of Aarhus, Denmark. From 2000 to 2004 he held a position as Assistant Professor at BRICS, University of Aarhus. Since 2004 he has been an Associate Professor at the Department of Computer Science, University of Aarhus. He has been affiliated the the Bioinformatics Research Center (BiRC) since its inception in 2001, and been its director since 2005.

Gerth Stølting Brodal received his Master's degree in 1994 and his PhD in 1997, both from the Department of Computer Science, University of Aarhus, Denmark. From 1997 to 1998 he was a post doc. in the group of Kurt Mehlhorn at the Max-Planck-Institute for Computer Science in Saarbrcken, Germany. From 1998-2006 he was affiliated with BRICS, University of Aarhus. Since 2004 he is an Associate Professor at the Department of Computer Science, University of Aarhus.

Rolf Fagerberg received his Master's degree in 1992 and his PhD in 1997, both from Department of Mathematics and Computer Science at the University of Southern Denmark. From 1998 to 1999 he was a post doc. at the same department. From 1999 to 2004 he held assistant and associate professorships at Department of Computer Science, University of Aarhus, Denmark. Since 2004 he has been an associate professor at Department of Mathematics and Computer Science at the University of Southern Denmark.