An $O(n^2 + \text{output})$ algorithm for reporting all shared quartets between two binary trees

Thomas Mailund

January 21, 2007

I describe an algorithm that in time $O(n^2 + \text{output})$ reports all quartet topologies shared between two binary trees. The algorithm uses a dynamic programming approach similar to Christiansen et al.[2] to efficiently collect intersections of sets of leaves of pairs of sub-trees from the two input trees, and then use the resulting sets to enumerate all shared quartets in constant time per quartet.

Keywords: Quartets; Tree comparison

Introduction

Efficient algorithms exist for calculating the quartet distance between pairs of trees—whether binary[1] or general trees[2,3]—but these algorithms only count the number of different or shared quartet topologies between two trees and do not explicitly report *which* quartets have different or equal topologies.

In the following, I present an algorithm that reports the set of quartets with the same topology in a pair of binary trees—of which, of course, there can be $O(n^4)$—in time $O(n^2 + \text{output})$ where n is the size of the set of leaves in each tree (this set is assumed to be the same in the two trees). The algorithm follows the same form as the $O(n^2d^2)$ counting algorithm in Christiansen et al.[2]: in a preprocessing step it iterates through all pairs of edges, one from each tree, and build a data structure representing the shared quartets between the two trees. In Christiansen et al.[2] this was only the size of intersections of sets; in the new algorithm it is an explicit representation of the intersection sets. Using dynamic programming, the structure is build in $O(1)$ time for each pair of edges. Following this preprocessing step, all shared quartets can be reported in time $O(1)$ per reported quartet.

The algorithm

The key observation used in Christiansen et al.[2] is that by orienting the edges in the trees, each butterfly quartet is associated with exactly one edge, and for binary trees where all quartet topologies are butterfly topologies. By rooting the two trees in an arbitrary leaf—but the same leaf in the two trees—we implicitly orient all edges, say towards or away from the root. To each edge we then associate the quartets with one leaf from each of the two sub-trees rooted in the source of the edge and two leaves selected outside these two sub-trees.

Let e be an edge (oriented) in tree T and let A and B be the sets of leaves in each of the two sub-trees behind e and C the set of leaves in the sub-tree in front of e (see Fig. 1). The quartets uniquely associated with edge e are the quartets $ab|cd$ such that $a \in A$, $b \in B$, and $c, d \in C$. Similarly, let e' be an edge in T'. and A' and B' be the set of leaves in the two sub-trees behind e' and C' the leaves in the sub-tree in front of e'. The set of quartets $ab|cd$ shared between e and e' are then the quartets with $a \in A \cap A'$, $b \in B \cap B'$ and $c, d \in C \cap C'$ plus the quartets $a \in A \cap B'$, $b \in B \cap A'$ and $c, d \in C \cap C'$. If, for each pair (e, e') we can enumerate the elements in each of the sets $A \cap A'$, $A \cap B'$, $B \cap B'$, $B \cap A'$, and $C \cap C'$ in time proportional to the size of the set, we can output all quartets shared between e and e' in time proportional to the output. We can then trivially iterate through all pairs of edges and output all shared quartets between the two trees in time $O(n^2 + \text{output})$. The crux of the algorithm is building a representation of these sets that lets us do exactly that, in time $O(n^2)$.

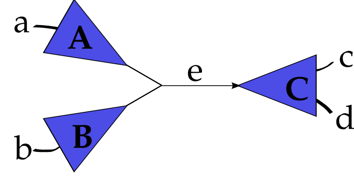

Fig. 1: Quartets $ab|cd$ claimed by oriented edge e.

Preprocessing

In the preprocessing step of the algorithm first root both trees in an arbitrary leaf and then build two tables containing intersection sets: one, $\mathcal{T}_1$, containing the intersection sets of trees pointing towards the root and one, $\mathcal{T}_2$, containing the intersection of trees containing the root. Both tables are indexed by pairs of (non-oriented) edges, e, e'; implicitly the edges are considered oriented towards the root for table $\mathcal{T}_1$ and away from the root for $\mathcal{T}_2$.

We first consider $\mathcal{T}_1$. For this table, edges are implicitly oriented towards the root, see Fig. 2. The source node of e (or e') is either a leaf or an inner node with two sub-trees. In the first case, the set of leaves beneath e is the singleton set corresponding to this leaf, in the second case, the set of leaves beneath e is $A \cup B$ where A and B are the sets beneath the children of e (see Fig. 2). Similarly for e' in T'.

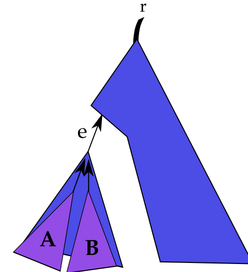

Fig. 2: The sub-tree beneath e when considered oriented towards the root.

This leaves is with the following cases:

1. e and e' both have a single leaf beneath them. $\mathcal{T}_1[e, e']$ should be set to $\emptyset$ if the leaves beneath the edges differs or the singleton if they are the same.
2. e has a single leaf beneath it, say a, and e' two trees with leaf sets A' and B' respectively. $\mathcal{T}_1[e, e']$ should be set to $(\{a\} \cap A) \cup (\{a\} \cap B')$.
3. e' has a single leaf beneath it, say a, and e two trees with leaf sets A and B respectively. $\mathcal{T}_1[e, e']$ should be set to $(\{a\} \cap A) \cup (\{a\} \cap B)$.
4. e has two trees with leaf sets A and B beneath it, and e' two trees with leaf sets A' and B'. $\mathcal{T}_1[e, e']$ should be set to $(A \cap A') \cup (A \cap B') \cup (B \cap A') \cup (B \cap B')$.

We will represent the sets in $\mathcal{T}_1$ as trees (with no degree-2 nodes) having the elements of the sets as leaves. We can compute these trees recursively. The basis case is item *1.* above; here we simply construct the empty tree or a tree with a single node as appropriate. Items *2.* and *3.* are treated similarly: If both the two sub-sets are empty, we insert $\emptyset$ into $\mathcal{T}_1$; if only one of the two sub-sets are empty, we insert the tree representation of the other in the table. Since a cannot be in both A and B—these sets are disjoint since the represent the leaves of two disjoint subtrees—these are the only options here. For item *4.* we build a tree for the four sub-sets as follows: If all four subsets are empty, we build the empty tree; if exactly one subset is non-empty, we insert that tree into the table; if two or more subsets are non-empty, we construct a new node and add the tree representations of the non-empty subsets as subtrees of this node.

Using memorisation or dynamic programming, we can ensure that all sub-trees are available when we need them in time $O(1)$ and by merging them together using pointers rather than copying sub-trees we can also perform the merging operations in time $O(1)$ leading to a total time for computing the full table of $O(n^2)$. For each entry in the table, the tree is of the same size as the set it represents and by traversing the tree we can enumerate all elements in the set in time proportional to the size of the set.

The table $\mathcal{T}_2$ is computed in exactly the same way as $\mathcal{T}_1$. The sub-trees behind the edges are still either singletons or an inner node with two subtrees, only now one of the sub-trees contain everything above the edge (towards the root) in the tree while the other is the sibling of the node pointed to by the edge, see Fig. 3. For the recursion, however, that is of no consequence.

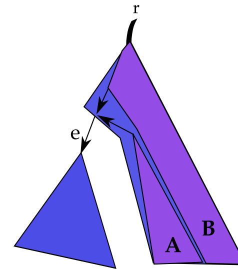

Fig. 3: The sub-tree beneath e when considered oriented away from the root.

Reporting shared quartets

As mentioned earlier, each quartet is associated with exactly one edge after the edges are oriented. Consequently, each shared quartet between the two trees is associated with exactly one edge-pair. Now, consider such an edge pair, e, e'. If the source node of either is a leaf, then that edge can claim no quartets and consequently the pair claim no shared edges. We therefore assume that both e and e' have an inner node as source and let f and g denote the edges pointing to this node in T, see Fig. 4, and let f' and g' denote the corresponding edges in T'.

A shared quartet, $ab|cd$ claimed by the edge pair e, e' has $a \in A \cap A'$, $b \in B \cap B'$ and $c, d \in C \cap C'$ or $a \in A \cap B'$, $b \in A \cap B'$ and $c, d \in C \cap C'$. We can look these sets up in our tables as $A \cap A' = \mathcal{T}_1[f, f']$, $B \cap B' = \mathcal{T}_1[g, g']$, $A \cap B' = \mathcal{T}_1[f, g']$, $B \cap A' = \mathcal{T}_1[g, f']$, and $C \cap C' = \mathcal{T}_2[e, e']$. Outputting the shared quartets is then

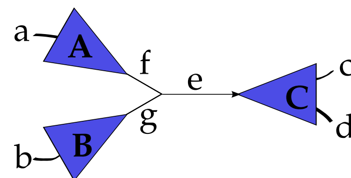

Fig. 4: Edge e with an inner node as source.

a simple matter of traversing the appropriate trees and outputting leaves.

In the reporting stage, we therefore spend time $O(n^2)$ for iterating through all pairs of edges, and spend $O(1)$ time for each reported quartet, giving us a total reporting time of $O(n^2 + \text{output})$.

Improvements

The algorithm can be improved in several directions:

1. Generalising to general trees. Reporting butterfly quartets in $O(n^2 d^2 + \text{output})$ for trees with max degree d is a trivial extension of the algorithm, but star quartets might be a problem.
2. Reporting the set of quartets that *differs* between two trees instead of those that are shared. If two trees are similar, this is more appropriate.
3. Construct a tree from the shared quartets without explicitly listing them; preferably in time $O(n^2)$ in total.

References

1. 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.
2. C. Christiansen, T. Mailund, C. N. S. Pedersen, and M. Randers. Computing the quartet distance between trees of arbitrary degree. In *WABI*, pages 77–88, 2005.
3. C. Christiansen, T. Mailund, C. N. Pedersen, M. Randers, and M. S. Stissing. Fast calculation of the quartet distance between trees of arbitrary degrees. *Algorithms Mol Biol*, 1:16, 2006. doi: 10.1186/1748-7188-1-16.