# Tree Distance

## Algorithms in BioInformatics 2 Mandatory Project 3

**Magnus Erik Hvass Pedersen (971055)**
**January 2005, Daimi, University of Aarhus**

## 1 Introduction

The purpose of this report is to verify attendance of the author to the *Algorithms in BioInformatics* course part II, at the department of computer science, University of Aarhus. The reader is assumed to be familiar with the problem description as well as the course litterature.

Following an outline of the problem that is to be solved, the solutions and their algorithms are described along with their implementations, after which experiments are conducted.

## 2 Tree Distance

Consider two unrooted trees $T$ and $S$ which have the same leafs but different topologies (that is, the trees have different inner nodes and connections amongst them). Assume the degree of each inner node is 3 or more, so that edges can not be prolonged with intermediate edges and nodes, more than what is needed in order to connect all of the leafs. This is not an essential assumption, and the methods described below, will work even though this is not the case. The assumption merely makes it easier to evaluate running time in terms of the number of nodes instead of the number of edges. Note also that leafs are unique within a single tree.

A distance measure between two such trees, is a metric allowing us to define and compare their similarity. However, not all valid metrics are useful, in that trees appearing to be quite similar, may exhibit maximum distance with some metrics.

### 2.1 Symmetric Distance

The *symmetric* distance is described in [1] and is easily understood, by considering all partitions of the leafs in the two trees $T$ and $S$, and then counting the ones that are unique – unique in the sense that they do not appear in the other tree. Note that two trees with a different number of leafs, will result in all partitions being unique. Also note that partitioning a tree so that it has one leaf in one partition, and the rest of the leafs in the other, is clearly not unique, as the exact same partitioning must exist in the other tree. However, this is not used in the present implementation.

As each leaf only occurs once within a tree (or more specifically, the species designated by a leaf, occurs only once within a single tree), then a partition of the leafs into two disjoint sets, is uniquely defined by one of the edges in the tree. Assume now that the two trees have only directional edges. That is, for two nodes $u$ and $v$ in $T$, if they are connected by an edge $e_{u,v}$ from $u$ to $v$, then they are also connected by another edge $e_{v,u}$ from $v$ to $u$.

Then define $k(e_{(u,v)}, f_{(u',v')})$ to be the number of common leafs, between the subtree of $T$ rooted in node $v$ and excluding the edge $e_{u,v}$ (and hence the subtree rooted in node $u$), and the subtree of $S$ that is rooted in $v'$ and excludes the edge $f_{u',v'}$.

Assume that all such $k$'s have been computed, then the symmetric distance (that is, the number of unique partitions in either of the trees), is a count over the $k$'s, in which the distance $d$ is initialized to zero, and then for each edge $e_{u,v}$ in $T$, $d$ is incremented by one if $k(e_{(u,v)}, f_{(u',v')}) + k(e_{(v,u)}, f_{(v',u')})$ does not equal $L$ (the number of leafs in a single tree), for each edge $f_{u',v'}$ in $S$. And likewise for each edge $f_{u',v'}$ in $S$, $d$ is incremented by one if $k(e_{(u,v)}, f_{(u',v')}) + k(e_{(v,u)}, f_{(v',u')})$ does not equal $L$ for each edge $e_{(u,v)}$ in $T$. Since this procedure counts each unique partition twice (both $e_{(u,v)}$ and $e_{(v,u)}$ are counted, and the same for edges in $S$), we must finally divide $d$ by 2.

### 2.1.1 Computing $k$ With Dynamic Programming

Clearly $k$ is a table of size $N \cdot M$ with $N$ being the number of edges in $T$ and $M$ the number of edges in $S$. If we start by computing the values that are easily determined, then we may be able to find recursions that will build the rest of the table in time $O(N \cdot M)$.

If we let $l$ be some leaf in $T$ and $l'$ a leaf in $S$, then $k(e_{(u,l)}, f_{(u',l')})$ is 1 if $l = l'$ (that is, if the two species designated by nodes $l$ and $l'$ are the same). On the other hand, if they are different, then set this value for $k$ to zero.

Let $k$ be shorthand for $k(e_{(u,l)}, f_{(u',l')})$, then it immediately follows that we may also set:

$$\begin{aligned}
k(e_{(l,u)}, f_{(l',u')}) &= L - 1 - (1 - k) \\
k(e_{(l,u)}, f_{(u',l')}) &= 1 - k \\
k(e_{(u,l)}, f_{(l',u')}) &= 1 - k
\end{aligned}$$

where $L$ is still the number of leafs in a tree.

Now to the rest of the $k$-table, where the idea is to use recursive traversal. That is, if we are to find the value $k(e_{(u,v)}, f_{(u',v')})$ and it has not already been computed, then if $v$ is not a leaf, find all the edges $e_{v,x}$ such that $x \neq u$ and let $k(e_{(u,v)}, f_{(u',v')})$ be the sum:

$$k(e_{(u,v)}, f_{(u',v')}) = \sum_{e_{(v,x)},\, x \neq u} k(e_{(v,x)}, f_{(u',v')})$$

If on the other hand $v$ was a leaf, then recurse through the $S$-tree instead:

$$k(e_{(u,v)}, f_{(u',v')}) = \sum_{f_{(v',x)},\, x \neq u'} k(e_{(u,v)}, f_{(v',x)})$$

At some point recursion will end, because when $v$ is a leaf we will start traversing $S$, and eventually reach a set of edges for which $k$ has already been calculated, either by previous recursions or during initialization. As each entry in the $k$-table is only traversed once, and it takes $O(1)$ time to compute a $k$-value from the $k$-values of its children nodes, then the total running time is $O(N \cdot M)$ – or as we have assumed that inner nodes have a degree of at least 3, then $N$ and $M$ are bounded by (a factor of)[1] the number of leafs $L$, so we have $O(L^2)$.

## 2.2 Quartet Distance

Let $e_{u,v}$ be an edge in $T$ that divides its leafs into two disjoint sets $A$ and $B$, and the edge $f_{u',v'}$ in $S$ does the same in $S$, but for the sets $A'$ and $B'$. Any two distinct leafs $a_i$ and $a_j$ from $A$, combined with any two distinct leafs $b_i$ and $b_j$ from $B$, form a socalled *quartet* $a_i a_j | b_i b_j$. The objective of the *quartet* distance, is then to count the number of unique quartets in the trees $T$ and $S$, that is, the number of quartets that do not appear in the other tree.

The quartet distance is described in [1] and an $O(L^2)$ algorithm is briefly discussed in [2], while a more elaborate description of that algorithm exists in [3, Chapter 3]. To use the notation of the latter, we wish to compute the quartet distance $d_Q$ given by:

$$d_Q(T,S) = |Q_T| + |Q_S| - 2 \cdot |Q_T \cap Q_S|$$

Which is similar to the (unwritten) formula for the symmetric distance, in that we take the total number of quartets in both trees, and subtract those that they have in common. As the ones that are common of course occur in both trees, both occurences must be subtracted from the distance count. This leaves us with the number of unique quartets in either of the trees.

The approach discussed here, is somewhat similar to the computation of the symmetric distance above. Let the edges $e_{u,v}$ from $T$ and $f_{u',v'}$ from $S$ split those two trees in the sets of leafs $A$, $B$, $A'$, and $B'$ as described above. First define a count for each edge in a tree, which gives the number of leafs in its subtree, and denote the count by:

$$c_T(e_{u,v})$$

That is, the number of leafs in the subtree rooted in node $v$, and arising from the removal of edge $e_{u,v}$ in tree $T$. Clearly this count may be computed recursively, by setting $c_T(e_{u,l}) = 1$ for all leafs $l$, and recursively computed for the internal nodes as follows:

$$c_T(e_{u,v}) = \sum_{e_{(v,x)},\, x \neq u} c_T(e_{(v,x)})$$

[1]To be more precise, if each inner node has degree 3, then the first three leafs means there is one inner node and hence three edges. For each additional leaf, the number of internal nodes is incremented by one, and the number of edges by two. Hence, there is $(|L| - 3) \cdot 2 + 3$ edges in total.

Now the number of pairs $a_i a_j$ with $a_i$ and $a_j$ from $A$ (that is, the subtree in $T$ induced by edge $e_{u,v}$), that do not exist in $A'$ (that is, the subtree in $S$ induced by edge $f_{u',v'}$), must be:

$$\alpha_T(e_{(u,v)}, f_{(u',v')}) = c_T(e_{u,v}) \cdot (c_T(e_{u,v}) - 1 - k(e_{(u,v)}, f_{(u',v')}))$$

as each uniquely appearing leaf in $A$, can be paired with any one of the other leafs in $A$, to form a pair that does not exist in $A'$ from the other tree $S$. Similarly we may take:

$$\alpha_S(e_{(u,v)}, f_{(u',v')}) = c_S(f_{u',v'}) \cdot (c_S(e_{u',v'}) - 1 - k(e_{(u,v)}, f_{(u',v')}))$$

as the number of pairs that occur in $A'$ but not in $A$. The number of unique quartets from $T$ in this regard, is then:

$$\beta_T(e_{(u,v)}, f_{(u',v')}) = \alpha_T(e_{(u,v)}, f_{(u',v')}) \cdot c_T(e_{v,u}) \cdot (c_T(e_{v,u}) - 1)$$

and similarly for tree $S$ and $\alpha_S$:

$$\beta_S(e_{(u,v)}, f_{(u',v')}) = \alpha_S(e_{(u,v)}, f_{(u',v')}) \cdot c_S(f_{v',u'}) \cdot (c_S(e_{v',u'}) - 1)$$

The sum of these two is the number of unique quartets in $A|B$ compared to $A'|B'$, and is denoted:

$$\gamma(e_{(u,v)}, f_{(u',v')}) = \beta_T(e_{(u,v)}, f_{(u',v')}) + \beta_S(e_{(u,v)}, f_{(u',v')})$$

which is a table similar to the $k$-table, and naturally also computable in $O(L^2)$ time.

The question is then, how many unique quartets does an edge $e_{u,v}$ in $T$ induce altogether, and not just in regards to a single edge $f_{u',v'}$ in $S$. First off we should select the minimum $\gamma$ from the possible combinations of edges from $S$:

$$\min_{f_{(u',v')} \in S} \gamma(e_{(u,v)}, f_{(u',v')})$$

because we must use the combination of edges which yields the fewest unique quartets, so a quartet that is in fact not unique, does not get counted as being unique. The same is done for the edges in $S$, as follows:

$$\min_{e_{(u,v)} \in T} \gamma(e_{(u,v)}, f_{(u',v')})$$

Which is the number of unique quartets that each edge in $T$ and $S$ contribute. Alas, in contrast to the symmetric distance, some of the edge-induced partitions of the leafs, will cause some quartets to be counted more than once, if they occur in the sub-trees of several different edges.

Weeding out the redundant counts of quartets can be done by preprocessing of the trees. A quite elaborate method is given in [3], and since it is limited to internal nodes of degree tree, we need an even more elaborate method for the present exposition, as we do not have such a limitation. The general idea however, is to recursively let edges *claim* quartets in their subtrees, and then by appropriate set intersections (see [3, Table 3.1] for the degree 3 case) combine the results of their children-nodes. Since each quartet can only be claimed by one edge, then each unique quartet will only be counted once. Since a fixed amount of time is needed for each edge, and the set of claimed quartets is computed only once per edge, then this can be achieved within the quadratic time bounds.

## 3 Implementation

Implementation is carried out in *MS Visual C++ .NET* and compiles to an *MS Windows* executable, with the entry-point located in the file `bioinf_distance.cpp`.

The class `LSpeciesTree` holds a species-tree (i.e. an object is instantiated for $T$, and another for $S$), and provides a number of functions to build the tree as well as accessing nodes and edges. The class `LTreeDistance` was thought as a base-class for computing in kind of distance between two evolutionary trees, but here it implements the symmetric distance. Its function `GetDistance()` is recursive and calls the `GetDistanceSum()` function in a `LSpeciesTree`-object, which then calls back into the `LTreeDistance`-object as needed. This mutual recursiveness and keeping track of which tree is being recursed, could be avoided by implementing the trees directly in `LTreeDistance`.

### 3.1 Tree Builder

The *Newick*-parser is taken from [4] and modified. First off, the file `lexer.cc` needed the following outcommenting and definition:

```
//#include <unistd.h>
#define YY_NEVER_INTERACTIVE 1
```

and `parser.cc` in the line:

```
extern const char *yytext;
```

had to have the `const` keyword removed, for it to be able to link the code. Apart from this, those two files are unchanged.

As the internal representation of trees in `LSpeciesTree` is based on integer-id's for nodes and edges, and pointers to the node- and edge-objects are never used outside the `LSpeciesTree`-class, the class `TreeBuilder` had to be rewritten to support this. The subclass of `TreeBuilder` for building an `LSpeciesTree`-object, is `LSpeciesTreeBuilder` which then implements the pure virtual functions accordingly.

Naturally, `LSpeciesTree` could be rewritten to use pointers to node- and edge-objects instead of integer id's, which would probably simplify the source-code a tad. The reason for using integer id's was to be consistent with the enumeration in the $k$-table.

## 3.2 Testing

Testing was performed in steps during development. Although socalled *assertions* were used to catch erroneous function-arguments and the like during debugging, the main issue with the algorithm for computing the symmetric distance between two trees, lies in the recursion of the $k$-table. The call-stack rapidly overflows if the algorithm does not terminate, but instead recurses indefinitely. Testing was focused on the computation of the tree-distance, and the Newick-parser was not explicitly tested.

To investigate errors in different phases of recursion, the example from [1, Figure 30.7, p. 529] was used, and `GetDistance()` was called with specific edges as parameters. As this was before the Newick-parser had been added, the trees had to be written in hand as follows. First the leafs are added, with the id's being identical for the two trees:

```
LSpeciesTree T, S;   // The two trees from figure 30.7
int u, v;            // Temporary nodes.

for (int i=0; i<7; i++)
{
    T.AddLeaf(i);
    S.AddLeaf(i);
}
```

Then tree $T$ is built as follows:

```
u = T.AddNode(0);
v = T.AddNode(u);
T.AddEdges(v, 3);
T.AddEdges(v, 5);
u = T.AddNode(u);
v = T.AddNode(u);
T.AddEdges(v, 4);
T.AddEdges(v, 6);
u = T.AddNode(u);
T.AddEdges(u, 1);
T.AddEdges(u, 2);
```

And the tree $S$ is built by the following:

```
u = S.AddNode(0);
S.AddEdges(u, 3);
u = S.AddNode(u);
S.AddEdges(u, 5);
u = S.AddNode(u);
S.AddEdges(u, 4);
S.AddEdges(u, 6);
u = S.AddNode(u);
```

```
    S.AddEdges(u, 1);
    S.AddEdges(u, 2);
```

After all errors had been resolved, the result of computing the symmetric distance for these trees and with this implementation, was 3 as expected.

Another test-example with only four leafs, two internal nodes, and 5 edges, was built as follows:

```
    u = T.AddNode(0);
    T.AddEdges(u, 1);
    u = T.AddNode(u);
    T.AddEdges(u, 2);
    T.AddEdges(u, 3);
```

for the $T$-tree. And for the $S$-tree:

```
    u = S.AddNode(0);
    S.AddEdges(u, 1);
    u = S.AddNode(u);
    S.AddEdges(u, 2);
    S.AddEdges(u, 3);
```

## 4 Experimental Results

The project description calls for experiments with Newick trees from the first mandatory project (on neighbour joining). It is thereby assumed that two implementations were made, one naive and one efficient, which were again assumed to result in different output, given the same similarity-matrix for a set of species. However, only one algorithm was implemented by this author, because it merely made use of clever data-structures to improve time-performance, and so, even if the naive implementation had been made, the output of the two would have been identical.

Having been unable to find other Newick-formatted trees, these experiments will have to be conducted with artificial data. To validate the correctness of the implementation, this is perhaps even better than real data, but to be able to evaluate whether the symmetric distance measure performs according to expectations, we would of course need real data.

The first pair of trees are written in Newick format as follows:

```
    (a, (b, (c, (d, (e, f)))));
```

and by interchanging `b` and `c` we get:

```
    (a, (c, (b, (d, (e, f)))));
```

for which the expected symmetric distance is clearly 2, which is also reported by the implemented program.

Now we three Newick formatted trees, let us call them $T_1$, $T_2$, and $T_3$, defined in that order as follows:

```
(a, (b, (c, (d, (e, (f, (g, (h, (i, j)))))))));
```

and $T_2$ being the interchange of `b` and `c` as above:

```
(a, (c, (b, (d, (e, (f, (g, (h, (i, j)))))))));
```

and $T_3$ having several changes:

```
(a, (c, (d, (b, (e, (h, (g, (i, (f, j)))))))));
```

Clearly the symmetric distance between $T_1$ and $T_2$ is 2 again, as the first two leafs are merely interchanged. This distance is also reported by the program. The distance between $T_1$ and $T_3$ can be easily counted after drawing the trees by hand, and is found to be 10. The distance between $T_2$ and $T_3$ is found the same way, and is 8 – both of which correspond with the output of the program.

As the above trees are as linear as possible, let us try comparing the first tree which was given above as:

```
(a, (b, (c, (d, (e, f)))));
```

with a star-like tree-topology:

```
((a,b), ((c,d), (e,f)));
```

The first tree has a unique partition given by `abc|def`, and the second star-like tree has a unique partition given by `abfe|cd`. That is, we expect the output of the program to be a symmetric distance of 2, which is indeed the case.

Trying the same thing with $T_1$, we may obtain the following tree $T_4$:

```
((((a,b), (c,d)), (e,f)), ((g, h), (i, j)));
```

where a drawing of the two trees, reveals the symmetric distance between $T_1$ and $T_4$ to be 6, which is also the output of the program.

Curiously a typo had first snuck in, so the expression was written as `.. (e,f))), ..`, and was therefore being parsed wrong. However, no exception was raised, nor was any error-message printed. Debugging the session revealed that there were 10 leafs in each of the trees, so the parser had seemingly not cut off any of the leafs. The parser however, did not raise a `Parser::ParseError` exception even though the parentheses were unbalanced.[2]

[2]Naturally, this is not critical in this application, as it is merely meant to demonstrate the algorithm for the symmetric distance, and the rest of the program is also not safe-guarded with consistent exception-handling.

## References

[1] J. Felsenstein. Inferring Phylogenies, Sinauer Associates, 2004

[2] D. Bryant, J. Tsang, P.E. Kearney, and M. Li. Computing the quartet distance between evolutionary trees. Proceedings of the 11th Symposium on Discrete Algorithms (SODA), pp. 285-286, 2000.

[3] John Tsang. An Approximation Algorithm for Character Compatibility and Fast Quartet-based Phylogenetic Tree Comparison. M.Sc. Thesis. University of Waterloo, Ontario, Canada, 2000.

[4] Thomas Mailund, Christian Storm. QDist version 1.0.3, `http://www.birc.dk/Software/QDist/index.html`