Neighbour Joining

Algorithms in BioInformatics 2
Mandatory Project 1

Magnus Erik Hvass Pedersen (971055)
November 2004, 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, and [1] in particular. Following an outline of the algorithm that is to be realized, a detailed description of a direct but still efficient implementation is given.

The actual programming is then described, including some modifications to source-code from the previous projects of this course. Finally, experimental results indicate the correctness and performance characteristics of the program.

2 Neighbour Joining

An abstract algorithm for socalled *neighbour joining* is found in [1, p. 171]. It operates on two sets L and T in which nodes represent species whose relationship in terms of evolutionary distance, is to be found.

Initially, both L and T hold all n species. In T there are no connections amongst their respective nodes, but in L they are all connected to each other, with the connections being weighted by the distance between the respective nodes. The idea is then to iteratively add fewest possible intermediate nodes to T, causing all its species to be connected in a rooted tree-like structure.

This is done iteratively by finding the pair of nodes i and j in L whose distance D_{ij} is minimal. A new intermediate node k is then created and added to T, and connected to nodes i and j with appropriate distances. The two nodes i and j are then removed from L, and a sibling of node k is added, for which the distance to all other nodes $m \in L$ is also set appropriately.

When there are only two nodes left in L, their counterparts in T are simply connected with their distance d_{ij} in L.

2.1 Distance

Note that there are two distances in play, d_{ij} and D_{ij}, where the distance between species i and j is given by d_{ij}, and are read from a matrix in socalled *Phylip* format. The formulae for calculating the new distances d_{ik} for some new node k, are found in [1] and are not required to understand the data-structures

and inner workings of the implementation. The distances D_{ij} equals d_{ij} minus the average distance between nodes i and j, and all the other nodes in L.

2.2 Sets

Making an efficient implementation of the above algorithm, requires a few data-structures to allow for quick look-up of various distances and auxiliary values, and a natural form is therefore to consider the two sets L and T as the main constructs.

2.2.1 T-Set

The simplest of these data-structures is perhaps the one for the set T, in which we need to add the nodes representing the species, and then add intermediate nodes to connect them. An efficient way of adding a new node to the set T, and connecting it to two other nodes $i, j \in T$, is to use an incremental index for each node, and then storing the nodes in an array. Since the algorithm adds one node to T in each iteration and there are $n-2$ such iterations, then we may allocate the array beforehand to avoid reallocation. The nodes are therefore both added and found in constant time.

The connections of nodes in T are stored in the data-structures for the nodes themselves, and by using a linked list, we may also achieve constant-time insertions here. When the distance-tree is output, the connections will be traversed from one end, and no other access to the connections is required – in particular, random access to a node's connections is not needed.

2.2.2 L-Set

In the set L, all nodes are connected to each other, again with the same distance in each direction, and the distance from a node to itself being zero. The set can therefore be implemented as a symmetric matrix with zero diagonal, that is, we only need to store e.g. the lower-triangular part (without the diagonal), and just make sure the look-up functions take this into account.

Adding or removing nodes from L, then corresponds to adding or removing rows and columns of the distance-matrix. To avoid the expensive operation of reallocating the matrix, and since two nodes are removed and one is added for each iteration, we simply keep an array of what nodes are active, and which rows and columns in the distance matrix they are associated with. The array therefore maps a row-number (or equivalently a column-number) to the id of the node that occupies it, and -1 if unused.

Another array stores the sum of all distances from each node $i \in L$ to all other nodes m in L:

$$\sum_{m \in L} d_{im}$$

which must be updated when a node is added or removed from L, taking time $O(n)$. Since $D_{ij} = D_{ji}$ we only need to compute and compare the values for

e.g. $i > j$, which takes time $O(n^2)$. As this must be performed for each of the $n-2$ iterations it takes to create the set T, the complete time complexity for the algorithm becomes $O(n^3)$.

This overall time-usage could probably be lowered by re-arranging the distance-matrix when a node is removed, so we would not have to go through all nodes in each iteration, as more and more nodes become unused. Secondly, it appears possible to keep a sorted data-structure of what distances D_{ij} are lowest, and update it when removing and adding nodes, thus lowering the $O(n^2)$ time-usage for finding the minimum distance.

2.3 Tree-Root

The tree that is built in T, is not unique in regards to its root, as any intermediate node may be selected for this. In this implementation, the root of T is set to the last added intermediate node. However, if the nodes are only connected in one direction, this does not guarantee that all other nodes can be reached from the root, and we must therefore connect nodes in T in both directions.

3 Implementation

Implementation is done in *Microsoft Visual C++ .NET* and compiles to an *MS Windows* executable.

The implementation of the score-matrix from the previous projects in part I of this course, has been revised to handle floating point scores (here, distances), and names that are strings instead of just single characters. It is still found in `LScoreMatrix` but is now a template class, thus allowing for the name- and number-types to be user-defined. This unfortunately requires the function-bodies to be implemented in the header-file also.

The classes for the sets L and T are named `LSetL` and `LSetT` respectively, and `LNodeT` is then the base-class for nodes in T, which is sub-classed to `LLeafT`, holding the given species' name.

During development, a small hand-written distance-matrix with only 3 species was used to uncover errors in the program.

4 Experimental Results

A selection of distance-matrices originating from [2] were used in these experiments. The one found in the problem-description is filed under the name `Acyl_transf_2.phylip`, and the output of the neighbour joining program for this matrix, is as follows, in socalled *Newick* format:

```
((((LXD2_PHOLU:0.0336604,((LXD1_PHOLU:0.0048875,
Q56818/8-3:0.0019125):0.0261214,
Q93CP5/8-3:0.0351086):0.0258646):0.112422,
LUXD_VIBHA:0.144383):0.0368528,Q9S3Z2/8-3:0.115141):0.0175141,
```

```
((LXD2_PHOLE:0.084609,LXD1_PHOLE:0.088861):0.0328969,
LUXD_PHOPO:0.0980531):0.0623672,Q9AJA7/8-3:0.120455);
```

which can be depicted as in figure 1, where the names have the sub-string `/8-3` removed, for [3] to accept the data.

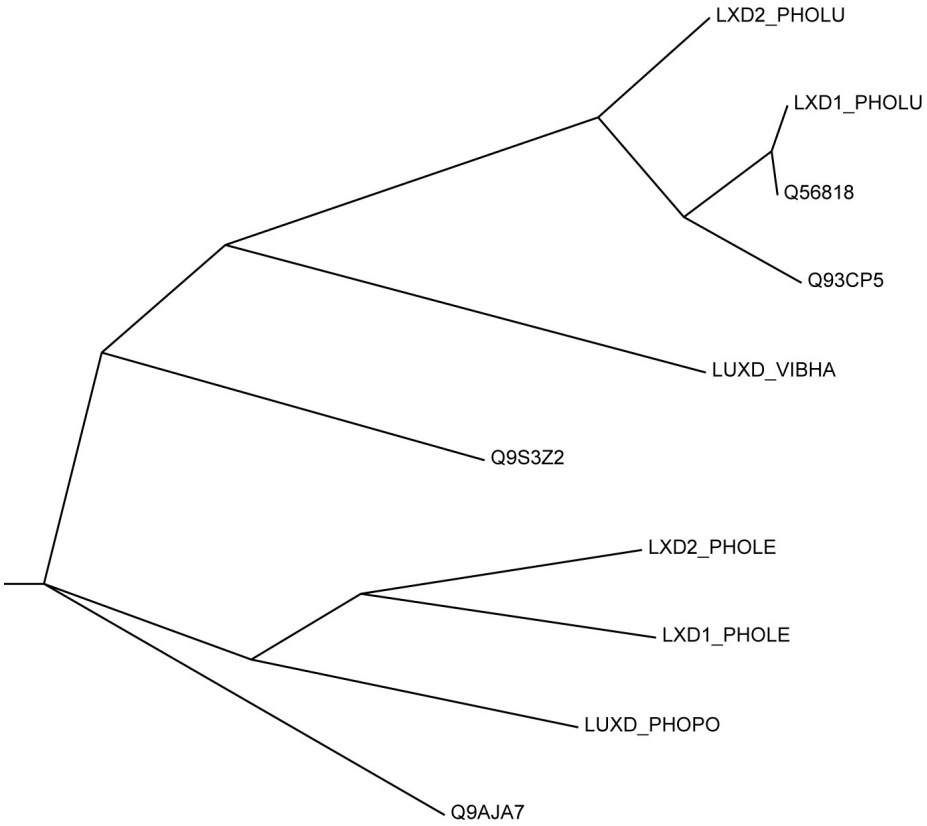

Figure 1: The evolutionary tree for the species in `Acyl_transf_2.phylip`.

Measuring the time it takes to actually process L and create the intermediate nodes in T (as done by the function `LSetL::Process()`), we arrive at figure 2 for various distance-matrices. As can be seen, the measure is sometimes locally erratic, but the overall tendency is cubic, so the number of milli-seconds spent on creating the distance-tree for a matrix with n species, is approximately:

$$msecs(n) \simeq \alpha \cdot n^3$$

where $\alpha = 51264/2991^3 \simeq 1.916e-6$ is found from the largest of the used distance-matrices, having $n = 2991$ species and using 51264 msecs. This relation corresponds well with the numbers in figure 2, and shows that the time complexity is as expected.

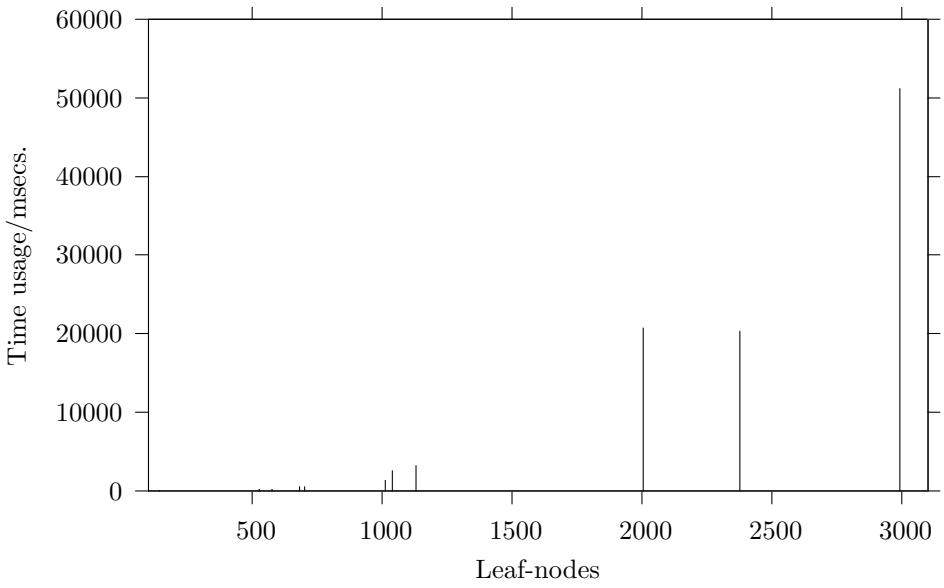

Figure 2: Time usage for various distance matrices.

References

[1] R. Durbin, S. Eddy, A. Krogh and G. Mitchison. Biological Sequence Analysis: Probabilistic models of proteins and nucleic acids, Cambridge University Press, 1998

[2] Protein families database of alignments and HMMs. `http://www.sanger.ac.uk/Software/Pfam/`

[3] Phylodendron, Phylogenetic tree printer. `http://iubio.bio.indiana.edu/treeapp/treeprint-form.html`