Programming Lab I
Handout 5

Dr. Martin Vogt
martin.vogt@bit.uni-bonn.de

May 19, 2020

Deadline: Jun 7, 2020 **Next Handout: Jun 2, 2020**

Phylogenetic Trees – Hierarchical Clustering

Motivation:
Evolutionary relationships between organisms might be estimated on the basis of homologous sequences and appropriate sequence alignment. As a quantitative measure for distance one can consider the number of mutations that have occured between the two sequences. Ignoring gaps, one can determine the so called fractional alignment distance, commonly called p-distance. If two sequences have L aligned positions of which D have different bases/amino acids the p-distance is

$$p = \frac{D}{L} = 1 - s$$

where s is the percentage sequence identity. p itself is not a good measure for the number of mutations itself assuming that mutations happen randomly at each position of the sequence and mutations can happen at the same position repeatedly over time. A simple correction is to model frequency of mutations at a site using Poisson distributions, which leads to

$$d = -\log(1-p) = -\log(s)$$

Using these distance values evolutionary relationships can be described using phylogenetic trees. An obvious way to generate such trees is to use hierarchical clustering.

In this week's exercise we would like to implement the basic agglomerative hierarchical clustering algorithm. The idea is simple and can be described in a few steps:

1. Start by assigning each object to its own cluster.
2. Select the two clusters that are closest to each other.
3. Merge the two clusters.
4. Repeat steps 2 and 3 until only one cluster is left.

The second step requires further specification. In order to specify closeness the concept of distance between individual objects needs to be extended to distance between clusters of objects. This can be done in a number of different ways, for phylogenetic trees UPGMA or average clustering is popular:

Choose the average distance between objects of the two clusters, i.e.

$$d_{\mathrm{AL}}(A,B) = \frac{1}{|A||B|}\sum_{a\in A}\sum_{b\in B} d(a,b)$$

More formally, the algorithm can be described as follows:

function HierCluster(n, D) ▷ Cluster n objects with pairwise distances D
  Let F be the set of all current clusters.
  Initially, F consists of all n one-element clusters C_0.
  Each cluster is associated with a height h.
  The height of all initial one-element clusters v is $h(C_0)$=0.
  while There is more than one cluster in F **do**
    Find the two closest clusters C_1, C_2 according to D
    Merge C_1 and C_2 by forming a new cluster C with children C_1 and C_2.
    Set the height of C to $D(C_1, C_2)/2$
    Determine the distance of the new cluster C to all clusters in $F \setminus \{C_1, C_2\}$
    Remove C_1 and C_2 from F and add C to F
    (Remove the distances from and to C_1 and C_2 from D.)
  return final vertex in F

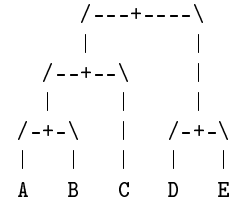
Exercises

Ex.1 *(4 pts) Reading in distance matrices*
Assume that all the distances have already been calculated and are stored in a text file similar to the Blosum matrices of the previous weeks. Two files, one containing pairwise distances between 5 objects (`small-distances.txt`) and one containing pairwise distances betwenn 13 objects (`distances.txt`) are given. Write a function that is able to read distance matrices and store them, for instance, in a dictionary of dictionaries so that distances can be accessed like `dist['D']['B']`.

A central question when implementing the algorithm is how to represent the tree-like cluster structure. This can be done in a surprisingly easy fashion, by representing clusters by nested tuples. The one-element clusters are simply represented by their identifiers as strings from the distance matrix and larger clusters are represented by nested tuples, which will reflect the tree-like structure:

A key operation of the hierarchical clustering algorithm is to take two clusters and merge them to a new cluster. The merging of two clusters can be represented by a tuple. E.g., consider objects `'A'`, `'B'`, `'C'`, `'D'`, and `'E'`. Then, `('A','B')` and `('D','E')` are two clusters obtained by joining `'A'` and `'B'` in the first case and `'D'` and `'E'` in the second case. Now consider merging `('A','B')` with `'C'`, which yields the three element cluster `(('A','B'),'C')`. Finally this cluster is merged with `('D','E')` to yield `((('A','B'),'C'),('D','E'))`.

Notice how the structure of the tree is reflected in the nested parenthesis structure.[1] From the final cluster `((('A','B'),'C'),('D','E'))`, the tree can be reconstructed:



Because clusters are represented as tuples they can now also be used as keys and we can use them to store the distances between clusters, e.g. `dist[('D','E')][(('A','B'),'C')]` represents the distance between clusters `('D','E')` and `(('A','B'),'C')` stored in a dictionary of dictionaries. In the following, distances are assumed to be stored in this way and the data structure is simply referred to as 'distance matrix'.

The most complex part of implementing the algorithm consists of updating the distance matrix. To this end new keys have to be inserted into the distance matrix and their distance to the other clusters has to be determined.

[1] In fact, there is a one-to-one correspondence between all possible rooted binary trees and correctly parenthesized expressions.

Ex.2 *(4 pts) Number of elements of a nested tuple*
First, write a function that counts the number of elementary objects in a nested tuple. I.e., the function should return 3 for `(('A','B'),'C')` and 5 for `((('A','B'),'C'),('D','E'))`. This function will be helpful when determining cluster distances.

Ex.3 *(4 pts) Merging clusters*
When two clusters are merged the distance of the merged cluster to all other clusters has to be determined. Given two clusters R and S that are merged to a new cluster $M = (R, S)$ the distance of M to a cluster T can be determined using

$$d(M,T) = \frac{1}{|R| + |S|} \left(|R|d(R,T) + |S|d(S,T)\right)$$

Write a function taking three parameters: a distance matrix (i.e. a dictionary of dictionaries as in exercise 1) and two clusters (represented as strings/tuples) that merges two clusters by updating the distance matrix.

Note, that after merging clusters R and S to cluster $M = (R, S)$ the clusters R and S are no longer needed. Their keys should be removed from the distance matrix. You can use `del dist[key]` to remove a key from a dictionary. To remove R, for instance, you not only need to remove `dist[`R`]` but also `dist[`T`][`R`]` for all other clusters T.

Ex.4 *(4 pts) Find closest clusters*
Write a function that takes a distance matrix as input and returns the two clusters that should be merged, i.e. whose distance is smallest.

Ex.5 *(4 pts) Hierarchical clustering*
Write a function implementing the hierarchical clustering according to the pseudocode. The function should return the final clustering as a tuple and the heights for each cluster. The height should be stored as a dictionary, where the key is the cluster and the value the height. Test your program using the two files `small-distances.txt` and `distances.txt`. To visualize the result you can use the function `showtree` provided in `showtree.py` by copying that file from the workshop folder and using `from showtree import showtree`.

Ex.6 *(Optional: 3 pts) Object oriented approach*
The proposed implementation represents the trees, which in turn represent clusters, as nested tuples. However, Python is (also) an object-oriented language and it seems a good idea to represent clusters using classes. Rewrite the hierarchical clustering code by representing clusters/trees as classes with a set of appropriate methods. To get you started see, for instance, `https://stackoverflow.com/questions/36263402/tree-class-implementation-with-node-and-leaf` for an implementation of a tree data structure using a base class and two sub-classes representing either inner nodes or leaves of the tree. For your class you might want to have methods/fields for returning the height or the number of elements, i.e. leaves.

You can still use a dictionary for all pairwise distances. However, you will need to implement the methods `__hash__` and `__eq__` for your classes. see, for instance, `https://stackoverflow.com/questions/4901815/object-of-custom-type-as-dictionary-key`. (Note, that your class should be *immutable*.)

Note, that the modified versions of the clustering functions from exercises 3–5 can remain global functions and do not (or should not) be methods of the class representing the tree. The tree class should just replace the tuple representation and the dictionary keeping track of the heights.

Ex.7 *(Optional: 3 pts) Plotting the clustering tree*
Use a general purpose plotting library like matplotlib or ASCII-art, as illustrated on the second page to visualize the clustering as a tree. (For the ASCII art the heights need not be correct.) There are packages available in Python for plotting trees and graphs. Try to avoid these and think of an algorithm for displaying hierarchical trees!