Computational Biology
Tufts University Bio 35
Fall 2021

Assignment 8: Human evolution

Human evolution and the UPGMA algorithm: All of your code for this problem should be in a file called `HumanEvolution.py`. At the end you will also make a text file with comments called `HumanEvolution.txt`. This assignment is taken (with minor adaptations) from the companion material for CFB. You can find the original at `https://www.cs.hmc.edu/twiki/bin/view/CFB/HumanEvolution`.

Introduction

Biological sequences can provide valuable information to help us understand human evolution. In this problem we will use information from mitochondrial DNA sequences to help us understand the origin of modern humans, and their relationship with Neanderthals. To do this we will implement the UPGMA algorithm for phylogenetic reconstruction.

Need to write the phylogenetic drawing tools so they can use them, since I have not assigned them.

Getting started

You can begin by creating `HumanEvolution.py`. We will be re-using code from the `DrawTrees2.py` and `PhylogeneticTrees.py` assignments, so you should begin by putting these files in the directory you are working in. Then put:

power example

```python
from DrawTrees2 import *
from PhylogeneticTrees import *
```

at the top of your `HumanEvolution.py` file. Next find the following `mitoData.py` file in the scripts directory for this assignment and put it in the directory you are working in. Add the following to the top of your `HumanEvolution.py` file:

power example

```python
from mitoData import *
```

This will import the data contained in the file into your environment.

This file contains two data sets related to human evolution, as well as several test sets. The human evolution related data includes fossil hominins (Neanderthal, and a modern human fossil from Kostenki Siberia), as well as data from several modern human groups. The data is based on whole mitochondrial genomes, excluding one particular region called the D-loop region. Based on this sequence, distances have been calculated between pairs of species/groups.

Neanderthal data

This data is based on mitochondrial sequence from the following:

Data table	
Samples	Notes
Chimpanzee	common Chimpanzee
San	San individual from southern Africa
Yoruba	Yoruba individual from western Africa
Finnish	Finnish individual from northern Europe
Neanderthal	38,000-year-old fossil of Neanderthal individual from Croatia
Kostenki	30,000-year-old fossil of individual from Russia with modern human morphology

`neandList` and `neandMatrix` are defined in the `mitoData.py` file.

power example
```
>>> from mitoData import *
>>> neandList
[('Chimpanzee', (), ()), ('Neanderthal', (), ()), ('San', (), ()),
    ('Yoruba', (), ()), ('Finnish', (), ()), ('Kostenki', (), ())]
>>> neandMatrix
{(('San', (), ()), ('Yoruba', (), ())): 0.005386, (('Chimpanzee', (), ()),
   ('Neanderthal', (), ())): 0.088209 ...
```

`neandList` is a list of the species and/or human groups. Each element in that list is actually a tiny tree with just that species/group (e.g. `('Chimpanzee', (), ())`). Recall from our discussion in the text that we always want to operate on trees: trees come into our program and a tree comes out of our program!

`neandMatrix` is a dictionary containing the distances between all pairs of species/groups. (In this example, the dictionary is big so we've only shown a few entries.) For example, the distance from San to Yoruba is 0.005386, so there's an entry in the dictionary for that. Notice that the key in this entry is not `('San', 'Yoruba')` but rather `(('San', (), ()), ('Yoruba', (), ()))` - that is, it's the little San tree and the Yoruba tree. This convention will make things easier when we write our code.

Human data

A second data set, of the same structure, contains samples from modern humans.

Data table	
Samples	Notes
San	San individual from southern Africa
Yoruba	Yoruba individual from western Africa
Finnish	Finnish individual from northern Europe
Kikuyu	Kikuyu individual from eastern Africa
Papuan	Papuan individual from New Guinea
Han	Han individual from China

power example
```
>>> humansList
[('San', (), ()), ('Kikuyu', (), ()), ('Yoruba', (), ()),
    ('Han', (), ()), ('Finnish', (), ()), ('Papuan', (), ())]
>>> humansMatrix
{(('Han', (), ()), ('Papuan', (), ())): 0.001874,
    (('Kikuyu', (), ()), ('Kikuyu', (), ())): 0.0...
```

UPGMA

Next, you'll write the program to build phylogenetic trees from distance matrices and display them. Your program should have the functions below. You may add additional helper functions if you wish, but it's not necessary. In any case, test every function using small test inputs before you move on to the next function.

1. `findClosestPair( speciesList, Distances )`. This function takes as input a species list (a list of trees represented as tuples) and a distance dictionary and returns the pair of different trees in the `speciesList` that have the least distance between them. Let's say we have two trees from `speciesList` called `treei` and `treej`. Remember that to get the distance between them, you can access the `Distances` dictionary like this:

   power example
   ```
   Distances[(treei, treej)]
   ```

   `findClosestPair` should consider all possible pairs of trees in speciesList, and return the pair with the minimum distance as a tuple (e.g. if `treei` and `treej` happened to be the closest pair, we would return `(treei,treej)`.

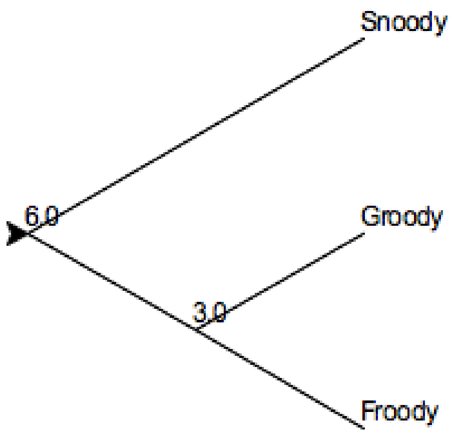
2. `updateDist( speciesList, Distances, newTree )`. This function takes the current list of trees called `speciesList`, the `Distances` dictionary, and the new tree that we have just built by merging the closest pair of species (as found by `findClosestPair`). This function does not return anything, but it does modify the `Distances` dictionary. Note that the two species trees that were merged are assumed to have been removed from the `speciesList` already but `newTree` has not yet been added to that `speciesList`. This function changes the Distances dictionary by adding the distances between `newTree` and all of the other trees in `speciesList`. This function is short and sweet (about 8-10 lines of code suffice!). Here are the parts:
   - The `newTree` is the merger of two trees. We can easily get these trees and give them names; let's call them `treeI` and `treeJ`.
   - The distance between `newTree` and any other tree, call it `treeK`, in the `speciesList` is computed as follows (note that we wrote a function `leafCount` in the `PhylogeneticTrees.py` homework): `leafCount(treeI)/ (leafCount(treeI) + leafCount(treeJ))` * (the distance from `treeI` to `treeK`) + `leafCount(treeJ)/ (leafCount(treeI) + leafCount(treeJ))` * (the distance from `treeJ` to `treeK`). (Remember to multiply integers by 1.0 before dividing in order to make sure that you don't get integer division unintentionally!)
3. `upgma( speciesList, Distances )`. This function takes as input the `speciesList` and the `Distances` dictionary and runs the algorithm that we described in the text. It repeatedly does the following until there is only one tree in the `speciesList`, at which point that phylogenetic tree is returned.
   - Find the closest pair of species in the `speciesList` (using `findClosestPair`).
   - Merge these two species trees into a new tree. The root of that tree is a number which is half the distance between the two species trees being merged.
   - Update the `speciesList` by removing the two species. The function `speciesList.remove(blah)` will remove the item named `blah` from the list named `speciesList`.
   - Update the distance dictionary by calling `updateDist`.
   - Add the new merged tree into the `speciesList`.
4. `drawPhyloTree2`. You can use the `drawPhyloTree2` function you've previously written to display these trees. `drawPhyloTree2` finds the height value at each internal node, and writes that on the tree it's drawing. If the number it finds there is a very long decimal, then `drawPhyloTree2` will write the whole thing. When you get numbers with many decimal places, as we will here, this can make the diagram harder to read. If you like, you can slightly modify your `drawPhyloTree2` function to fix this. We had been converting numbers to strings using the built-in `str()` function. Instead, we can use a function called format. Format allows us to take a decimal number, round it to some number of decimal places, and convert it to a string. For example, this syntax will round to three decimal places:

   power example
   ```
   >>> format(.987564,".3f")
   '0.988'
   ```

Test cases

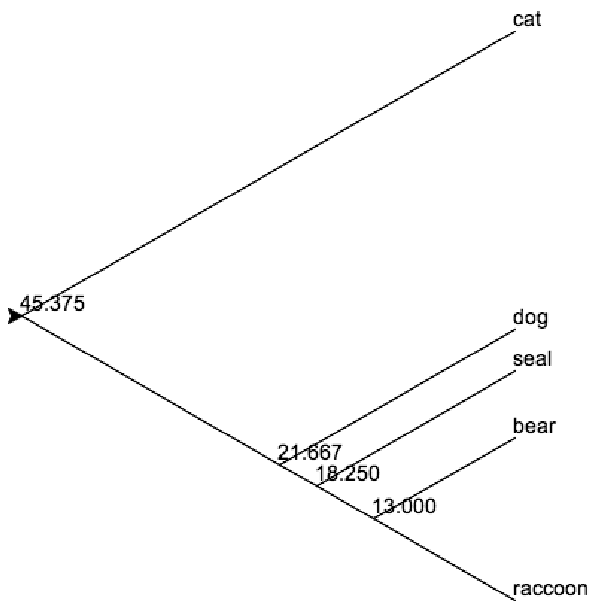
Some examples of our code running on test data:

power example
```
>>> groodiesTree=upgma(groodiesList, groodiesMatrix)
>>> drawPhyloTree2(groodiesTree,25)
```



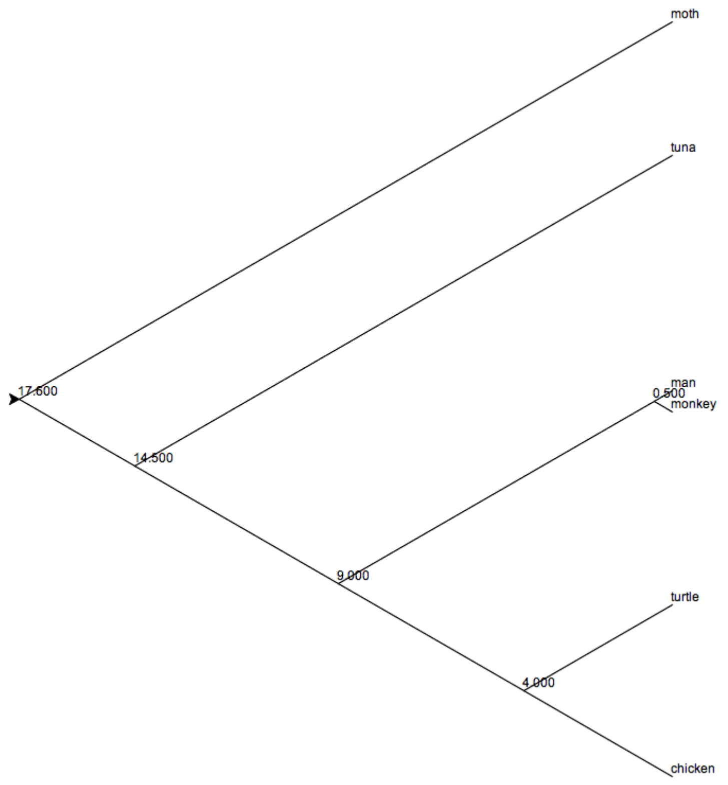
power example

```
>>> carnivoresTree=upgma(carnivoresList, carnivoresMatrix)
>>> drawPhyloTree2(carnivoresTree,6.5)
```



power example

```
>>> fitchTree=upgma(fitchList, fitchMatrix)
>>> drawPhyloTree2(fitchTree,35)
```



Answering questions about human evolution

Use `upgma` to generate trees for the two main data sets:

power example
```
>>> neandTree=upgma(neandList, neandMatrix)
>>> humansTree=upgma(humansList, humansMatrix)
```

Next calibrate them assuming a chimpanzee-human divergence of 6 million years. (You wrote the scale function in the `PhylogeneticTrees.py` assignment.)

power example
```
>>> neandTreeScaled=scale(neandTree,6.0/neandTree[0])
>>> humansTreeScaled=scale(humansTree,6.0/neandTree[0])
```

Then plot the scaled trees with `drawPhyloTree2`, and answer the following questions in your `HumanEvolution.txt` file:

- Are Neanderthals more closely related to modern Europeans than to other modern humans? Does it appear that modern Europeans descended from Neanderthals?
- Does this data support an African origin for modern humans?
- Compare the divergence time of Neanderthals from modern humans with that among modern human groups. What is the ratio between these two?