Algorithms in Bioinformatics
Project 5 - NJ tree construction

Ditte Torlyn (201408508)
Rune Wind (201608439)
Astrid Christiansen (201404423)

April 2020

Introduction

In this project, we implement an algorithm for neighbor-joining (NJ). We compare the performance of our implementation to the NJ programs QuickTree and RapidNJ.

Implementation

How we have implemented the NJ algorithm

Our implementation of the NJ algorithm is written in Python. We have implemented the algorithm as described on slide 62 about tree construction. In the algorithm, we iteratively want to find the 2 points that are closest to each other while also being furthest away from the rest of the points. We approximate this by finding the points i and j that minimize

$$n_{ij} = d_{ij} - (r_i + r_j)$$

where d_{ij} is the (i, j)'th value from the distance matrix and $r_k = \frac{1}{|S|-2} \sum_{m \in S} d_{im}$.

In our algorithm, we make a list of leaf names and their corresponding indices in the distance matrix:

```
S_nodes = letters
S = list(range(len(letters)))
```

where `letters` are the letters from the distance matrix file. Then, while we have more than 3 nodes left, we do the following.

We make a list of the r_k's, so that we do not have to calculate the same values multiple times, but can simply look them up:

```
row_sum_list = [1/(len(S)-2) * sum(dist_matrix[i][:]) for i in S]
```

Then, instead of filling out the entire N matrix, we only fill out the upper right triangle in N, since N is symmetric, so this is all of N that we need (we simply put 0's in all other entries):

```
N = [[dist_matrix[i][j] - (row_sum_list[i] + row_sum_list[j])
    if j > i else 0 for j in S] for i in S]
```

Then we find the minimum value in N and its indices (i, j):

```
N_upper = [N[i][j] for i in S for j in S if j > i]
min_val = min(N_upper)
min_i_j = np.where(N == min_val)
listOfCordinates = list(zip(min_i_j[0], min_i_j[1]))
min_coord = None
for coord in listOfCordinates:
    if(coord[0] != coord[1]):
        min_coord = coord
        break
i = int(min_coord[0])
j = int(min_coord[1])
```

When we create the new node k, we use our `row_sum_list` to look up the r_k's:

```
# Calculate weights and find leaf names
weight_ki = (1/2) * (dist_matrix[i][j] + row_sum_list[i] - row_sum_list[j])
weight_kj = (1/2) * (dist_matrix[i][j] + row_sum_list[j] - row_sum_list[i])
leaf_1 = S_nodes[i]
leaf_2 = S_nodes[j]

# Create Newick formatted node
k = "(" + leaf_1 + ":" + str(round(weight_ki, 3)) +
    ", " + leaf_2 + ":" + str(round(weight_kj, 3)) + ")"
```

Then we update the `dist_matrix` by removing row i and j and column i and j, and adding a new row and column for node k:

```
# Create row for new node k to insert
row = [(1/2) * (dist_matrix[i][m] + dist_matrix[j][m] -
dist_matrix[i][j]) for m in S if m != i and m != j]

# Remove row i and j
dist_matrix = np.delete(dist_matrix, [i,j], 0)
```

```
# Remove column i and j
dist_matrix = np.delete(dist_matrix, [i,j], 1)

# Insert row and column for new node k
dist_matrix = np.row_stack((dist_matrix, row))
dist_matrix = np.column_stack((dist_matrix, np.append(row, 0)))
```

We then update S:

```
S_nodes.remove(leaf_1)
S_nodes.remove(leaf_2)
S_nodes.append(k)
S = list(range(len(S_nodes)))
```

We continue to do these steps until we have only 3 nodes left in S. When we have only 3 nodes left, we add the last 3 weights to the tree:

```
i, j, m = S[0], S[1], S[2]
weight_vi = (1/2) * (dist_matrix[i][j] + dist_matrix[i][m] -
            dist_matrix[j][m])
weight_vj = (1/2) * (dist_matrix[i][j] + dist_matrix[j][m] -
            dist_matrix[i][m])
weight_vm = (1/2) * (dist_matrix[i][m] + dist_matrix[j][m] -
            dist_matrix[i][j])
```

We write the last part of the Newick tree and return the tree string:

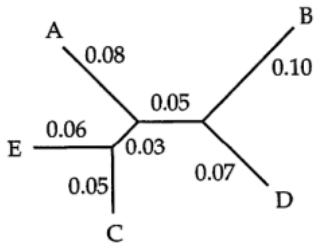
```
v = "(" + S_nodes[i] + ":" + str(round(weight_vi, 3)) + ", " +
    S_nodes[j] + ":" + str(round(weight_vj, 3)) + ", " +
    S_nodes[m] + ":" + str(round(weight_vm, 3)) + ");"
return v
```

Status of our work

Our program works correctly on the test case in `example_slide4.phy`:

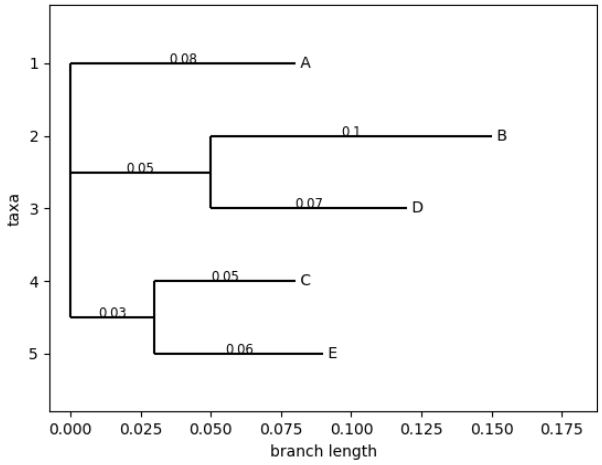
```
5
A 0.00 0.23 0.16 0.20 0.17
B 0.23 0.00 0.23 0.17 0.24
C 0.16 0.23 0.00 0.20 0.11
D 0.20 0.17 0.20 0.00 0.21
E 0.17 0.24 0.11 0.21 0.00
```

i.e. we get the tree on slide 4:



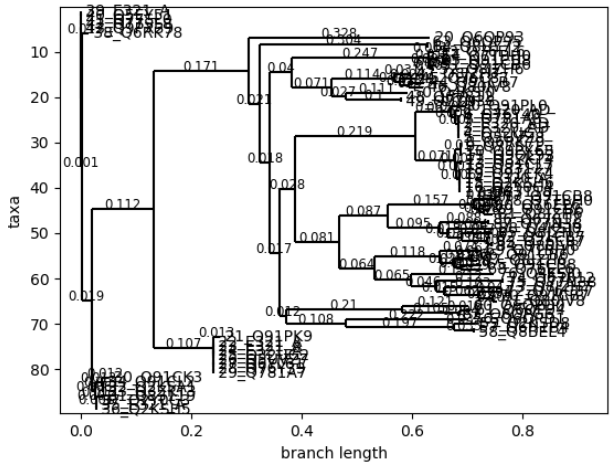
Our program creates the following tree in Newick format:
`(A:0.08, (B:0.1, D:0.07):0.05, (C:0.05, E:0.06):0.03);`

When we draw our tree using the `Phylo.draw()` function, it looks like this:



We see that our tree corresponds to the tree from slide 4.

For bigger files, we have used the online tool Phyfi at `https://services.birc.au.dk/phyfi/go.py` to visualize our trees instead of the `Phylo.draw()` function, as the trees become quite big. For example, here is the constructed tree from the file `89_Adeno_E3_CR1` from `distance_matrices.zip` when drawn using `Phylo.draw()`:



How to run our program

Our program file is called `nj.py`. Running this program will construct an unrooted tree from a given dissimilariry matrix. To run our program from the command line, type:

```
python nj.py matrix.phy
```

where `matrix.phy` is a file specifying the dissimilarity matrix in Phylip format.

The output will be a tree in Newick format and it is written to a file named `tree.new`.

Environment

We performed all our experiments on a machine with the following specifications:
– Intel Core i7 2.6 GHz (Turbo Boost up to 4.3 GHz) with 9 MB cache
– 16 GB RAM
– macOS Mojave (V. 10.14.6)

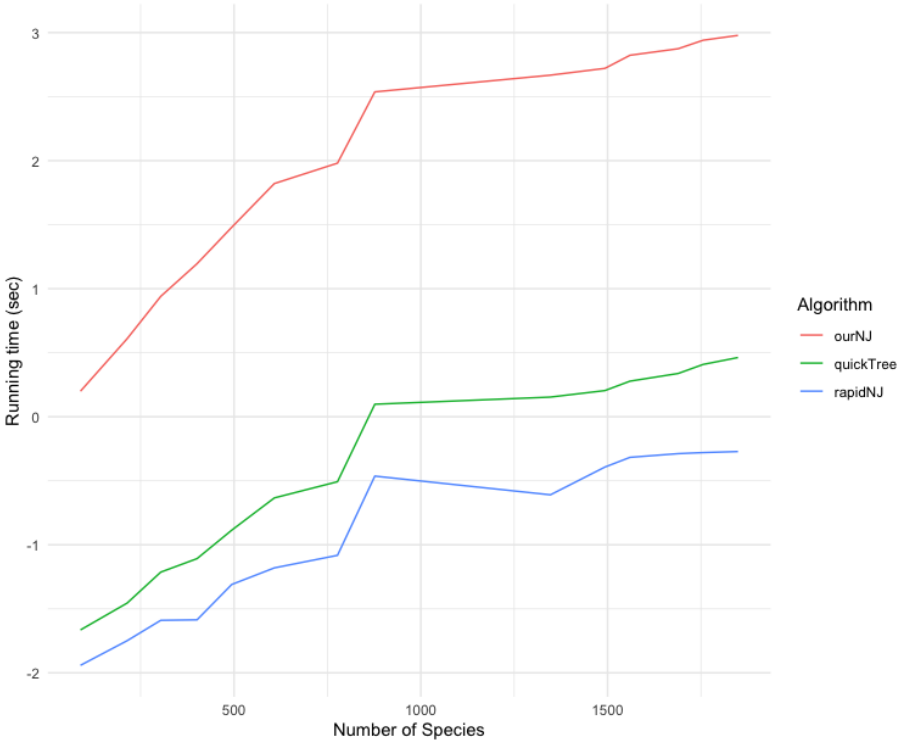
Our program is written in Python. We timed the running times of the programs using the `time()` function from the `time` library. The experiment was performed by looping through the input files and running each of the three tree construction algorithms. The measured time included reading input and writing output.

Experiments

We have created a table with a row for each of the 14 distance matrices in `distance_matrices.zip`. The table contains the running times of QuickTree, RapidNJ and our NJ algorithm in seconds for each distance matrix. It also conatins our speed-up relative to QuickTree and RapidNJ, and lastly it contains the RF-distances between the trees constructed by the 3 algorithms. The RF-distances were calculated using a script supplied by Christian Storm from BiRC at Aarhus Univeristy. In the table, we annotate QuickTree as QT, RapidNJ as RNJ and our implementation as NJ.

	Time (s)			Speed-up of NJ		RF-dist		
Size	QT	RNJ	NJ	QT	RNJ	NJ & QT	NJ & RNJ	QT & RNJ
89	0.022	0.011	1.58	0.014	0.0072	16	32	30
214	0.035	0.018	4.07	0.009	0.0044	18	52	56
304	0.061	0.026	8.71	0.007	0.0030	12	78	78
401	0.078	0.026	15.66	0.005	0.0017	32	88	98
494	0.130	0.049	30.18	0.004	0.0016	203	503	532
608	0.232	0.066	66.23	0.004	0.0010	20	12	20
777	0.311	0.083	95.61	0.003	0.0009	208	476	494
877	1.252	0.344	344.46	0.004	0.0010	32	44	54
1347	1.424	0.246	465.27	0.003	0.0005	2	4	2
1493	1.601	0.405	525.76	0.003	0.0008	34	98	92
1560	1.896	0.482	655.39	0.003	0.0007	76	154	152
1689	2.177	0.515	748.35	0.003	0.0007	38	104	102
1756	2.563	0.525	872.35	0.003	0.0006	22	78	76
1849	2.899	0.535	951.02	0.003	0.0006	98	278	280

As seen in the table, we did not get a speed-up compared to QuickTree and RapidNJ. As seen by the RF-distances in the table, we construct trees that are relatively similar to the trees constructed by QuickTree, compared to those constructed by RapidNJ. This makes sense, since QuickTree implements the basic cubic time algorithm, as does our algorithm. Our algorithm is however slower than QuickTree, as we will also illustrate with the following graph.



In the graph, we see that our algorithm is much slower than both QuickTree and RapidNJ. This is a bit weird, since we have implemented approximately the same algorithm as QuickTree, but maybe it just comes down to the used programming language. Since QuickTree is written in C and our algorithm is written in Python, is makes sense that QuickTree is faster than our algorithm.