Final Project Rough Draft

Problem: The maximum parsimony problem takes as input a set of n strings, each string a sequence of k nucleotides (A, C, T, or G). These strings have names, as in s1, s2, s3, …, sn. The objective is a tree on these strings that has the optimal parsimony score.

Input: The input will consist of an some number pairs of strings, each with one string of length 3 (identifiers) and one string of some length k (sequences). The data will contain n elements.

Example Input: where k=6, n=3

```
>SBC
CTAACA
>SBF
CCAATA
>SCD
CTATTG
```

Output: The output will be the bottom row of the tree given as an ASCII tree, newick structure, and the score of the tree

Example Output:

```
   /-SCD
--|
  |   /-SBC
   \-|
      \-SBF
(SCD,(SBC,SBF));
Cost: 5
```

We have coded two heuristics for finding the maximum parsimony of a tree. Both begin by generating G(V,E) where V is given by the sequences and E is given by the Hamming distance between any two nodes in V. The first heuristic is the 2-approximation heuristic given in class found by calculating the MST of G. This maximum parsimony approximation tree (MPAT) is then constructed from a linear ordering of the leaves if you span the nodes of the spanning tree. The second uses a MST on the leaf nodes and then subsequently on each row of internal nodes to generate an approximation tree. This creates a MPAT that does not have just a linear ordering of the leaves, but an ordering that is derived based on the MST of each set of internal nodes. (See *Appendix* for more information*.)* We know the first heuristic is a 2-approximation.

> ***Proof:*** *Let O be an optimal set of sequences. Double every edge in O, this graph has a Eulerian tour X, since every node has even degree.The Eulerian tour X includes internal vertices and leave of O, and cost(X) = 2*cost(O).By triangle inequality, cost(C)<=cost(X). Remove one edge from C to obtain a path P. Then P is a spanning tree on S, and cost(P)<=cost(C)<=cost(X)=2*cost(O)*
>
> *Now let T be a mst on S; then cost(T)<=cost(P)*

We expected to find that Heuristic2 gave a more optimal result because it does more work but this turned out not to be the case. Heuristic1 generates the estimate tree by picking a random node to be the root whereas Heuristic2 generates the estimate tree by assuming all given nodes to be leaves. It makes a less-than-optimal assumption that it should pair up as many of the given nodes as it can. In doing so, Heuristic2's tree makes inoptimal pairings. This can be seen in *Appendix Step 3* where 1 and 3 are paired, though it would have been optimal to pair 2 and 3. Heuristic1 properly pairs 2 and 3, generating a more optimal solution. Another primary roadblock in development of Heuristic2 was a poor estimation of ideal ancestor nodes. In order to properly estimate an ancestor that leads to a low cost, the heuristic would need to generate ancestor nodes(*1*) based on what its siblings are going to be. Because we did not find a good way to "predict the future", we simply set the ancestor of a pair to the same value as that of the left child. Given the issues we ran into generating Heuristic2, Heuristic1 will always generate an output with a score no greater than the score of Heuristic2.

If we were to further this research, we would find a way to attempt to “predict the future” to generate more optimal parent pairings. This would greatly reduce the cost of trees generated by Heuristic2, resulting in a result likely more optimal than Heuristic1.

Not only does Heuristic1 generate a better approximation, but its solution is generated in faster time. This can be shown by a simple running time analysis of the two algorithms. Heuristic one is $O(mlogm)$ where $m=|E|$ because it uses a comparison sort within Kruskal’s MST algorithm which runs in mlogm time. Heuristic two is $O(n^3)$ because we iterate through the vertices ($O(n)$), then within this iteration, we iterate through the edges of the calculated MST (Let $G'(V', E') = MST(G)$, thus $|E'|<=|V|$ thus $O(n)$), then within this iteration we again iterate through the edges of the MST ($O(n)$).

Runtime (in seconds)	**Heuristic 1**	**Heuristic 2**
Dataset 1 (25 sequences)	0.009914	0.016632
Dataset 2 (100 sequences)	0.165462	0.267023
Dataset 3 (3 sequences)	0.000164	0.000274
Dataset 4 (5 sequences)	0.000261	0.000484

MP-Cost	**Heuristic 1**	**Heuristic 2**
Dataset 1 (25 sequences)	550	593
Dataset 2 (100 sequences)	1649	1887
Dataset 3 (3 sequences)	5	5
Dataset 4 (5 sequences)	9	12

When choosing which heuristic to use in any given situation, a programmer must consider which he values most: short running time or more accurate approximation. In this case, Heuristic 1 is a better approximation for a programmer to choose since it is both faster and more accurate. Further improvements could be made to Heuristic 2 with a better ancestor prediction algorithm.

Souce code can be found at: https://github.com/jmadd/MaximumParsimony/

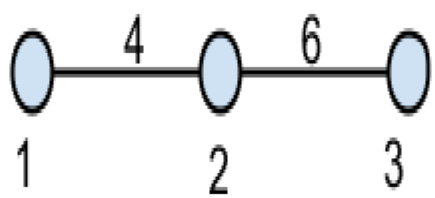
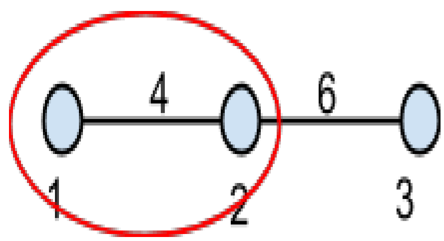
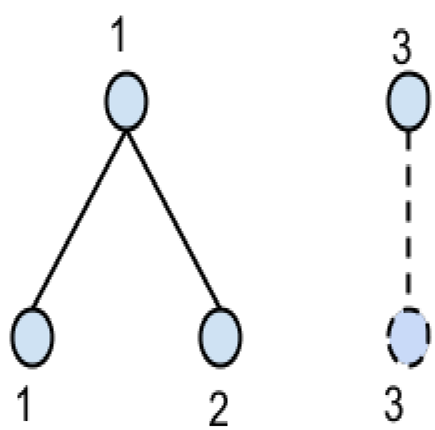
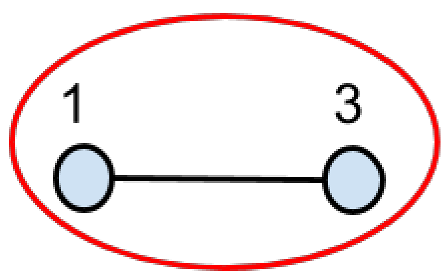
---Output Tree--Heuristic 1, 25 inputs---

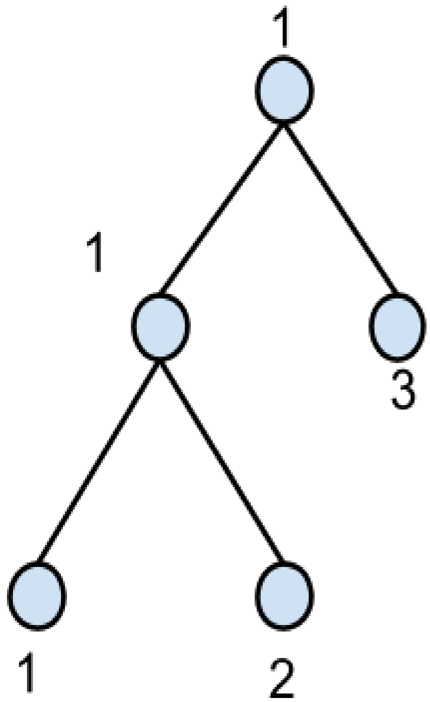
```
   /-SBC
  |
  |   /-SCD
  |--|
  |  |   /-SGD
  |   \-|
  |      \-SGF
  |
--|   /-SHG
  |  |
  |  |   /-SHH
  |  |  |
  |  |--|--SGH
  |  |  |
  |  |  |   /-SIH
  |  |   \-|
   \-|      \-SGC
     |
     |   /-SGE
     |  |
     |  |--SGJ
     |  |
     |  |   /-SFJ
     |  |--|
      \-|   \-SH
        |
        |   /-SBF
        |  |
        |  |   /-SHD
        |  |--|
         \-|   \-SHA
           |
           |   /-SEA
           |  |
            \-|   /-SII
              |  |
               \-|   /-SHI
                 |  |
                 |  |--SHB
                  \-|
                    |   /-SJH
                    |  |
                    |  |   /-SIB
                     \-|--|
                       |   \-SIG
                       |
                       |   /-SIJ
                        \-|
                           \-SGA
```

---Output Tree--Heuristic 2, 25 inputs---

```
      /-SGC
    /-|
   |   \-SIH
  /-|
 |  |   /-SGH
 |   \-|
 |      \-SHH
 |
 |           /-SGD
 |         /-|
 |        |   \-SGF
 |      /-|
 |     |  |   /-SBC
--|     |   \-|
 |   /-|      \-SCD
 |  |  |
 |  |  |   /-SGE
 |  |   \-|
 |  |     |   /-SFJ
 |  |      \-|
 |  |        |   /-SHG
 |  |         \-|
 |  |           |   /-SGJ
  \-|            \-|
    |               \-SH
    |
    |      /-SBF
    |   /-|
    |  |  |   /-SHA
    |  |   \-|
    |  |      \-SHD
    |  |
    |  |         /-SHB
     \-|       /-|
       |      |   \-SHI
       |   /-|
       |  |  |   /-SEA
       |  |   \-|
       |  |      \-SII
        \-|
          |      /-SIB
          |   /-|
          |  |   \-SIG
           \-|
             |   /-SIJ
              \-|
                |   /-SGA
                 \-|
                    \-SJH
---
```

Appendix (Heuristic2 explanation)

Step #	Image Representation	Explanation
1		Here is the MST for three nodes 1,2, and 3 with weights.
2		Now we select the cheapest edge of the MST to create an ancestor for nodes 1 and 2.
3		Here is our first level of internal nodes with 1 being passed up. Since 3 had no pair it is sent up to the next level of the tree.
4		We then find the MST between 1 and 3 which is just 1 and 3, and choose the only edge we can.

5		Finally we create the ancestor and pass up 1. Since there are no more nodes left on this level except 1, we are done and have the tree ((1,2),3) as our maximum parsimony approximation tree.

Sources

http://www.cs.utexas.edu/~tandy/331-approx-alg.pdf

```
def heuristic_1(V):
    tree = {}
    for i in V:
        final.append(i)
    V = range(0, len(final))
    head = 0
    G = fullyConnectedGraph(V)
    T = nx.minimum_spanning_tree(G)
    addToTree(tree, T, head)
    # adjust nodes to add transition of generations
    for node in tree:
        final.append(final[node])
        tree[node].append(node)
    return head, tree
```

```
def ancestor(s1, s2):
    return s1
def ObtainAnc(A, tree, V):
    for u, v in A:
        anc = ancestor(final[u], final[v])
        final.append(anc)
        tree[ len(final) - 1 ] = [u,v]
        V.append( len(final) - 1 )

def heuristic_2(V):
    tree = {}
    for i in V:
        final.append(i)
    V = range(0, len(final))
    head = None
    while len(V) > 1:
        G = fullyConnectedGraph(V)
        T = nx.minimum_spanning_tree(G)
        A = []
        while T.number_of_edges() > 0:
            u,v = cheapestEdge(T)
            A.append( (u,v) )
            if T.number_of_nodes() == 2:
                head = len(final)
            T.remove_node(u)
            T.remove_node(v)
        V = T.nodes()
        ObtainAnc(A, tree, V)
    return head, tree
```