CS170: Efficient Algorithms and Intractable Problems
Final Project

Algorithm Description: Minimum Routing Cost Spanning Tree Algorithm (Main):

Minimum routing cost spanning trees are spanning trees that minimize the pairwise distances between vertices. We derived our own algorithm to compute an MRCST from the inputs by learning and building on Rui Campos' algorithm that approximates the MRCST quickly. The algorithm uses Prim's algorithm as the basis but has two major changes. The first modification is that there is a deterministic way to select the starting vertex. It considers the degree of the vertex, the sum of adjacent edge weights, and maximum adjacent edge weight to compute the spanning potential of a vertex. It then selects the vertex with the highest spanning potential to be the root of the MRCST. Secondly, it uses different parameters other than the edge weights to select as the next edge in the MRCST. This algorithm computes a parameter that takes in the weight of a vertex and the estimated cost of the path between two vertices in order to select the next vertex. While the standard MRCST algorithm constructs a spanning tree, our output doesn't need to span every node in our input graph. Instead, we only need to span a dominating set of our original graph, thus we terminate our algorithm as soon as we reach a set of vertices that dominates our original graph G. While the MRCST algorithm produces the majority of our outputs, our actual output is the minimum of several of our earlier algorithms (detailed below) combined with our MRCST approximation.

Post Processing:

1. Node Expansion Optimization:

After receiving tree T which spans the nodes of G, we wanted to optimize our graph so that it includes any of the nodes in the set S := {G \ T} which decreased our pairwise distance average. Our node optimization algorithm iterates through all of the nodes in S and their edges and adds to our tree any that decrease our average.

2. Pruning:

After finding the spanning tree, the tree has additional vertices and edges that are necessary for a valid network. There could be potential gains from removing some of these edges and vertices. We developed two methods for pruning this tree for a more optimal result.

Pruning Algorithm 1:

Note: "Set" nodes cannot be pruned without invalidating the network.

1. For each leaf vertex in the tree
    a. Check to see if this leaf is "set". If it is, move on to the next leaf
    b. Check to see if removing this leaf will reduce the average pairwise distance of the network.
    c. If yes, remove this leaf and mark its parent as "set" (we cannot remove the parent or the network would cease to be valid)
2. Repeat step 1 until there is an iteration in which no leaves are pruned.

Pruning Algorithm 2:
The 2nd pruning algorithm first completely prunes the leaf nodes of the tree and adds the edges into a list. This list is then sorted by increasing weight. Then, for each edge, we compare if adding this edge will decrease the average pairwise distance compared to the completely pruned tree. If so, this edge is then added back to our tree.

Why do you think it is a good approach?

We believe that the MRCST algorithm was a good approach because it considered several factors in addition to edge weight to prioritize edges in our queue. Compared to Prim's and Kruskal's algorithm our MRCST was able to more accurately approximate a spanning tree with minimal pairwise distance. Additionally, we started from a vertex that had a high spanning potential and only expanded from there until we reached a dominating set of vertices. Intuitively starting from a high-degree low cost (edge weight) vertex is a good strategy to minimize the distance between any two nodes. Additionally, a greedy strategy allowed us to maintain an efficient runtime comparable to Prims. Though this only provides an approximation of the minimum pairwise distance dominating tree, our final outputs were quite close to the optimal leaderboard score, while running in O(nm + n^2/logn) time. We also combined this approach with several others, in order to maximize our results on graphs with varying topology and edge weight.

What other approaches did you try?
We used a series of algorithms in addition to our main algorithm. We discovered that, depending on the input, the algorithm derived from Campos's paper was not always the best. On certain inputs, other algorithms turned out to be better. To reflect this, for each input, we would run several different algorithms and take the best result from them.

Algorithms:
MRCST: Described above

Middle Expansion: This algorithm uses the same parameters as our MRCST algorithm to select a starting vertex V. It then sorts each other vertex, by its distance from V using V's shortest-path tree. We then expand a radius around V and add vertices to a set once they fall within the radius. We continue this process until our set dominates the original graph. We then use the MRCST to connect the vertices in our set.

Greedy Add: At each timestep find the vertex which increases our pairwise distance the least and greedily add this to our tree. Continue until we reach a dominating set. Preformed well on a significant number of inputs

MST: We used Prim's algorithm to construct the MST on our graph, and then used both of our pruning algorithms to reduce our tree to a dominating set.

Dominating Set: This finds the minimum dominating vertex set on the input graph as well as the minimum spanning tree. Then it uses a minimum set of edges from the minimum spanning tree to connect the dominating set.

Our final output is the tree which produces the minimum pairwise distance.

Resources:
We primarily tested our algorithms on our local computers. Occasionally, we ran our algorithm on a Google Cloud Compute Engine VM Instance when our algorithms were too slow. For the most part, we opted to run on our local computers for simplicity.