Adam Ovadia 12/26/2016

CSCI 335 Project 2

The three algorithms we implemented for the second project were to find an approximate solution to the traveling salesman problem. The traveling salesman problem is that you want to find a path from one location back to that same location while visiting all other locations on the way. The goal is to optimize this solution to find as short of a path as possible. The first algorithm was a greedy algorithm in which we find the smallest distance between two cities. Once we add that to the path, we find the next smallest distance. The difference here is that unlike the nearest insertion, we are not maintaining that the next smallest city will connect to a previous city in the path. This means that the second smallest distance (edge) may not be an edge connecting to a city already found (disjoint). This repeats for the total number of cities and for as long as no cycles are created. The second algorithm is the nearest insertion algorithm, in which we start out by finding the smallest distance between two cities. That is the starting path. Then find the next city closest to either the front or end of the path and add it to the path as long as no cycles are created. This repeats until all cities have been added. The third algorithm tries to improve the running time of the algorithm. The idea is to find the smallest edge or distance between two cities. Then repeatedly find the next smallest edge as long as they do not connect to any previously determined edges. Thus the number of edges will be half the number of cities. Then by comparing distances from edges to other edges, we combine the edges into a single path as long as no cycles are formed.

Analysis:

In the first algorithm, I was expecting a time complexity of O(n2), where n is the number of cities. Based on my implementation, I found this to be true. The main components of this time complexity in my code comes from computeEdges(), findDataPos(), and findSmallestEdge(). Both of these functions use nested for loops (2 for loops) which each have a complexity of O(n2). The overall complexity of O(3n2) is just O(n2). Other functions have either a constant time or a complexity of O(n), but n2 is the most significant.

In the second algorithm, nearest insertion, I was expected the time complexity of O(n2), where n is the number of cities. Based on my implementation of nearest insertion, I found this to be true. The main components of this comes from the functions, computePath() and computeAdjMatrix() because this function uses two for loops, which has a time complexity of O(n2). Other functions have either a constant time or a complexity of O(n), but n2 is the most significant. While this algorithm holds the same time complexity as the first algorithm, because the first algorithm uses more functions with an O(n2) time complexity, I expect and have confirmed through testing that this algorithm performs quicker than the first.

For the third algorithm, I assumed a time complexity of O(n2). The functions, computeEdges(), computePath(), connectEdgePath(), and findSmallestEdge() all have nested for loops (two for loops), which each have a time complexity of O(n2). The overall complexity of O(4n2) is just O(n2). Other functions have either a constant time or a complexity of O(n), but n2 is the most significant. Based on this, it should perform the slowest out of the three algorithms and by testing it has shown this property. This could be due to the implementation of the algorithm. My idea was that instead of having to continually check and recheck all edges to find out where each city belongs in the path, by using the already created weights to just create a vector of edges which would be only half the number of cities (each edge contains two cities). This way I would only have to determine how each edge connects instead of each city, since determine the edge costs is a common element in each algorithm.