Lab 3 WriteUp

## 

This table lists the results of the traveling salesman problem, listing the number of nodes, number of operations, minimum distance found, and theoretical number of operations, as tested on graphs of sizes between 4 and 10 nodes.

This graph compares the number of operations taken to check the Brute Force algorithm compared to the theoretical number of operations (nodeNum!). As seen, the number of operations perfectly matches the theoretical number, which is expected.

This graph shows the performance of my dynamic programming algorithm compared to the theoretical performance expectations. The lines do not match as with Brute Force, being closer to e2n rather than 2nxn2. The line has an almost perfect R2 value, so the trend is still exponential, just not following the theoretical expectations. Overall, the trend is similar, if a bit sharper.

This graph simply compares the performance difference between dynamic programming and brute force. As the number of nodes increases, the difference between the two increases rapidly.

The dynamic programming approach to the Travelling Salesman Problem, for a seemingly complex algorithm at first, is relatively simple. Assuming that the results depend on starting and ending on the same node, slowly break down the input vector, node by node, until there are no nodes left. For each break-down step, return the minimum distance from the current node to every other node remaining in the input vector, then remove the closest node. Repeat again with the new node, until there are no nodes left. For an example with 4 nodes, see below.

The function called to calculate the total path is a recursively calling function named “findPath”:

findPath(1, {2,3,4}, 1)

The first step of findPath is equivalent to the minimum of the following three statements

dist(1,2)+findPath(2, {3,4}, 1)

dist(1,3)+findPath(3, {2,4}, 1)

dist(1,4)+findPath(4, {2,3}, 1)

After calculating that minimum, the following step would be finding the minimum of the distance between the two remaining nodes, and then connecting to the last remaining node and travelling back to the start.

At first, my initial plan was to only implement a factory pattern, because they are fairly simple in creation, yet help organize the code without any major loss in efficiency like some other organization patterns sometimes do. It also makes object initialization incredibly easy, because one just has to pass the correct enum to the factory object. However, as I began working through the algorithms, I realized that a good chunk of the functionality could be abstracted behind a base class and turn the two algorithms into derived classes. While not the perfect representation of one, this turned the two algorithms into an implementation of a Strategy class. This will also allow easy adaptability for the addition of further algorithms in the future.