Post Lab 11

Gabriel Groover

**Implementation/Time Complexity**

For the prelab, I chose to build a new vertex class for each point of the graph. I then represented the graph as a vector of these vertex objects. The final topological sort was done with the STL queue. Each vertex contained a vector of other vertices to represent adjacent nodes, an int for indegree (used later to sort the graph) and a string to record the course the vertex represented. I used the vectors in each place because we were working with relatively small numbers of values and were unlikely to encounter the vector amortized time.

The first step in the topological sort is to read in the data. This will always be Θ(n) because we have to read in a set of n items. In order to add each element to the graph, I then had to loop through each vertex and adjacency list for the existing graph. This required a nested for loop and a running time Θ(n^2). I then use the same double for loop to actually add the new points to the graph. The linear time vector insertion is unlikely to occur in our purposes, and would not occur on every iteration and so our complexity is unchanged. To actually sort the graph, I create a queue from the vertices of indegree = 0. I then examine each element in this queue and loop through its adjacency list. This is once again Θ(n^2) so the total time complexity is Θ(n^2) for topological sort.

For the inlab I used the already existing middle earth implementation. I will not consider the creation of the earth or itinerary in my time analysis for this portion. I first use a string to keep track of the start city from the itinerary and remove this city from the itinerary vector to avoid changing the computed distance. Both of these operations are constant time, but unnecessary. I could've avoided this by simply using iterators to make next_permutation() and sort() skip the first element. In order to used next_permutation from the algorithm library we must first call sort on the vector which is Θ(nlog(n)). Next, I examine the distance for each path using next permutation. This will never be any running time besides Θ(n!). In order to compute the total distance of each one of these permutations, we must perform n-1 calls of getDistance between two cities. We must do this on each cycle of the while loop, so our final running time is Θ((n-1)n!). We can ignore the time required to print out the final route because it is negligible in comparison. However, for completeness I note that printRoute is linear time.

**Space Complexity**

I will now examine the space required for the data structures outlined above. Each vertex contains an int, string and a vector of vertex pointers. We will assume a max size of 8 bytes for the string representing the class index. This gives us a total vertex size of 4 + 8 + n(4) , or (12 + 4n) bytes. (n is the number of adjacent vertices) The only other data structures required to carry out the sort is vector<vertex*> and queue<vertex*>. The size of each of these is (m * 8) bytes, where m is the total number of graph nodes. If we do not consider temporary values used outside of the data structures, our topological sort has a space complexity of m(12 +4n) + 16m bytes.

For the traveling salesman I will assume the existence of a middle earth. We first generate an itinerary which consists of a vector of strings. The string representing each city will vary in length, but here I will use an average length of 8. The total size of our vector is then 8n where n is the length of our itinerary. I use a second vector to keep track of the minimum path that is also of size 8n. I use a float to

keep track of numeric minimum distance (to avoid future calculations) and a string to track start city that I discussed earlier. This makes the total space required for my salesman implementation 16n + 12 bytes.

**Traveling Salesperson Optimizations**

We can create an upper bounded approximation for the solution using minimum spanning trees. If you consider all of the cities as points, we can construct a minimum spanning tree where the root node is the start city. We can then simply traverse this new minimum spanning tree and use that as an approximation of our traveling salesman path. This will not always yield the exact solution, but it can never be more than twice the true solution cost. This is because the minimum spanning tree must be less costly than the traveling salesman path, else it wouldn't be minimum. The maximum cost that could be associated with a path is one that crosses every edge twice. The true value is between these two. This is a polynomial time algorithm because it requires no permutations and would be extremely fast/

If we were more concerned with the exactness of the solution, we could use what is known as iterative improvement. This is the idea of taking some, quickly obtained approximation and changing it multiple times to approach the true solution. If we consider an approximation as above, we can take a pair of two edges and remove them from the solution. In order to be a complete path, the nodes the edges connected must still connect. We then consider all other nodes to see if we can create a pair of edges that will reconnect those nodes in a shorter distance. We just keep doing this until it is no longer possible. The tradeoff is the time we lose to iterate through all the edge combinations. It is still vastly better than all the path permutations.

It is difficult to come up with an optimized exact solution because no real one exists. We can only achieve a slightly better exponential running time. One optimization that would increase speed is to throw out some path permutations that are not possible solutions. For example, if a pair of nodes is too much farther apart than the nodes that neighbor each one of the nodes, we can exclude permutations with that edge from our solution. If we can make this elimination with multiple pairs of nodes, we could save a lot of time. (still exponential)