1 Efficiency

1.1 Prelab

In the following evaluation, E will represent the number of edges in the graph and V will represent the number of vertices.

The first step in the code is to read in the graph. I use a map to store the nodes that have been created. For each edge I check whether there is an existing node for either end already. These checks have amortized constant running times. Next I loop through the vector of existing nodes to get pointers to each end of the edge. I then push the pointer to the target node to the source node's vector of adjacent nodes. Disregarding the constant timee operations, for each edge in the graph I iterate through all nodes in the graph once. This step thus has running time $\Theta(E \cdot V)$.

The actual sorting portion of the lab involves iterating through the nodes of the graph and calling visit on each unvisited node. Visit essentially performs a depth-first-search on the given node until it encounters nodes that it has already seen. The end result of this is that visit is called for each node and each edge in the graph. The running time is $\Theta(E + V)$.

Pointers to the V graph nodes are stored in a vector that takes $8 * V$. Each node itself has a vector containing pointers to adjacent nodes, a string containing its value, and a bit indicating its marked status. I assume that the strings representing classes have at most 8 characters. In the worst case there will be $V - 1$ adjacent nodes. Thus , a single node takes up approximately $8 * V + 10$ bytes. Additionally, a map is created with an entry for each node that is composed of a string and an int. There are V entries that each take approximately 14 bytes. Finally, another vector of Node pointers holds the final sorted order of the graph. This vector takes another $8 * V$ bytes.

1.2 Inlab

For a list of n cities there are $n!$ different ways of ordering them. Each of these possible combinations must be checked to see if it's the most efficient path. Checking an ordering involves iterating through the n nodes to calculate the length of the path.

My brute force solution to the traveling salesman problem has a running time of $\Theta(n! \cdot n)$ where n is the number of cities that are to be visited.

Including only data structures that are created as a part of the solution and not part of the problem (i.e. the MiddleEarth object) I used a single vector of city names that is permuted in place, a vector of names to store the best path, and an integer to store the shortest length. If cities are limited to names of 100 ascii characters then the total space taken is roughly $200 \cdot n + 4$ bytes.

2 Acceleration Techniques

My solution took approximately 19 seconds to compute the shortest tour of 10 cities. Using the brute force algorithm with running time $\Theta(n!)$ I can use this datapoint to estimate running times for various other solutions.

2.1 Dynamic Programming

The basic idea of dynamic programming is to avoid performing calculations multiple times. The dynamic programming acceleration rests on the obeservation that every subpath of a path of mimum distance itself has minimum distance. This means that we can calculate and store the solutions to every subproblem and then use those solutions to help solve the actual problem. For example if we wanted to calculate the shortest tour of 5 cities we would start by calculating all the shortest tours of one city. We then use those solutions to find the shortest tours of two cities, etc. This approach reduces the running time to $O(2^n n^2)$ at the cost of greatly increasing the amount of space used. I expect that finding a solution for 10 cities with this approach would take half a second.

2.2 Greedy Algorithm

This approach involves a greedy heuristic that gives an approximate solution. Beginning at some start node the algorithm simply selects the adjacent unvisited node that is closest to the current node until all nodes have been visited then returns to the start node. This results in a run time of $O(n^2)$. I estimate that this would run in five thousandths of a second.

2.3 2-opt

2-opt is a specific implementation of the general k-opt iterative strategy for improving a path. It works by removing intersections from the graph. Two edges are selected and removed from the graph, they're then replaced by two different edges that reconnect the graph. For example, if edges B-¿D and C-¿E are removed they might be replaced by edges B-¿E and C-¿D. This process is repeated until it no longer improves the path. This method is not guaranteed to arrive at the optimal solution but it often provides significant improvement over the starting path. As this is a method for improving an existing solution there is no real way to analyze its running time. The process can be aborted at any time and still result in an improved solution over what was started with (assuming that the original solution wasn't optimal).