Holt Crews, jhc6we, 12/1/17, postlab11.pdf

Prelab Analysis

In the prelab for topological sort, the time complexity depends on both the number of edges and the number of vertices. Each vertex must be iterated through and passed onto the queue at some point so the complexity for just the vertices is V. Also, every edge must also be handled as they determine the indegree of each vertex. This runtime is E. Therefore the total runtime of this algorithm for topological sort is (V+E).

The space complexity for topological sort usually is not very intense. The queue is continually having members added and deleted and doesn't typically take up that much space. However, it does have the potential to hold every node at once. This would not be a very useful graph as there would be no nodes but it is technically possible. In my implementation, I kept track of the names of the classes in a vector, having (V) complexity. A map of the string and its corresponding vertex, having (V) complexity. I kept track of pointers to each vertex, having (V) complexity. The graph itself has a vector of pointers to the vertices while the vertices have a vector of pointers to all of its adjacent vertices, in the worst case, every node could point to every other node and have a complexity of (V+E). The overall complexity is therefore (3V+V+E+V).

Inlab Analysis

As we know, the time complexity for the Traveling Salesperson problem is very intensive as it is an NP-complete problem. The main cause for this famously slow problem is the fact that every possible route has to be iterated through before one can be declared as the shortest. The permutations in this implementation of a brute force traveling salesperson problem have a complexity of (n-1)! where n is the total number of cities in the itinerary including the start point. My algorithm goes through each of these permutations and then calculates the distance of that iteration. Calculating this distance has a complexity of (n) because we have to visit every node. This constitutes the heavy lifting part of my code and it is only done once in order to save time. The total time complexity is therefore (n-1!*n) because each permutation has to run n times to calculate the distance. My printRoute utilizes the computeDistance function that creates a vector that is in the order of the shortest route.

While the time complexity may be enormous, the space complexity of this algorithm is relatively small. Besides keeping track of all of Middle-Earth, there is a vector of all of the cities which has a complexity of (n). And a temporary vector that keeps track of the shortest route up to that point, also complexity of (n). There are a few

other variables that are of negligible size and don't increase with the size of the problem. The total space complexity is thus (n+n).

Acceleration Techniques

1. **Nearest Neighbor Algorithm** - This algorithm can be described as a greedy algorithm, or one that takes whatever seems to be the best at any given time. These types of algorithms are often relatively efficient but many greedy algorithms can't guarantee that they return the optimal solution to a problem. The Nearest Neighbor algorithm starts by searching every adjacent node to the start node and takes the edge with the shortest distance. Then, from the next node, it takes the shortest edge coming from it. According to Wikipedia, this algorithm is on average 25% longer than the true optimal route. So, it usually doesn't return the shortest route but it will give you a reasonable route in most cases.
2. **Cutting Plane Method** - This exact algorithm solves the traveling salesperson problem through linear programming. The basic idea is to create inequalities around the solution space that further restrict and tighten the space. You start with a real solution space for the problem and then you create a series of inequalities that wraps around this space and includes all of this space so that the original space is kind of like a subset of this inequality space. This inequality space is much easier to compute optimal solutions for as it is more structured and can be solved using linear programming. Once you find an optimal solution in the inequality space and if that solution is not in the original problem space, you create a new inequality that sections off that optimal solution from the rest of the inequality space, so you are continually constricting this space more and more. At some point, you will get a solution for the inequality space that is within the original problem space and the first example of this is your optimal solution. With a little tinkering, this cutting plane method can be adapted for the Traveling salesperson problem.
3. **Ant Colony Optimization** - This heuristic algorithm is based off how ants lay out pheromone trails to show their fellow ants how to get to a food source. In the algorithm, an "ant" will go on a route partially determined by pheromone trails, when a decision needs to be made the ant is more likely to choose a path that has pheromones on it. Once the ant has completed the path to all of the required cities, if it is a reasonably short path, it will go back and mark that trail with pheromones to notify later ants. After a while, some of the older pheromone trails evaporate, leaving the more recent, more optimal trails. The reason this algorithm is a heuristic is because there are probabilities associated with whether or not an ant will take a pheromone-laden trail or a new one. These probabilities

have to be set by the creator and therefore there is some inherent error associated with this method.