//Yujian Li(yl7kd) Postlab 11 04/28/16 Section 102

PostLab 11

Time Complexity:

In the prelab topological.cpp, the time complexity can be analyzed into two sections. The first part consisted of creating the graph. I used the node and vector in order to represent the graph which means every strings in the file will be a node in the graph. At first it takes O(L) times to read all the input where L is the total amount of strings that are not equal to zero. Then I created two separated for loops inside to check if vector contains the string value. Which will also take O(L) runtime at the worst case. If the vector does not contain the string, then a new node will be created that contains that string value and pushed into the end of the vector which has a O(1) runtime and its adjacent vector which contains the other nodes that has relationship with the current node will push the other node into the vector with a O(1) runtime. The creating graph phase will then total result a O(L^2) runtime. The second part is the sorting phase. I used the for loop to first search through the nodes with a zero indegree with a runtime of O(n) then I push the node into the queue with a constant runtime O(1). Then if the queue is not empty, I will first save the front value of the queue in a node and pop the value. Then I loop through its adjacent vectors to see if any indegree equals zero after minus 1 then I will push that node into the queue again until the queue becomes empty which contains a nested loop that will result a O(L^2) runtime. Therefore, the total runtime of the topological.cpp runtime will be O(2(L^2)).

In the inlab section, the constructing a middle earth is one of the main component of the code. It will first uses a for loop to create a vector that contains C cities. Then it created a two D array that computes the distance position of each cities within the size of the world. In total it creates a O(C^2) runtime where C is the number of cities in the middleearth object. Then erases the first element of the object which takes a linear runtime O(C). The sort method will take a O((C-1)log(C-1)) runtime to prepare the permutation. Finally, the next_permutation method will generate all all the possible combinations within the values of the middle earth object which has a O((C-1)!)runtime. Inside it also invoke the computeDistance method. Because the getdistance method has a constant runtime O(1), the computedistance method will take overall O(C) times to get all the distance from the starting points to the end. Therefore the final runtime will be O(C*((C-1)!)).

Space Complexity:

In the prelab, when creating a node, it takes two unit space for indegree and string then an n space for vector which has a O(N) space complexity. Creating a graph which will take n unit space for the vecotor which contains n nodes. It has a space complexity of O(N^2). In the topological sort method, it will take a O(N^2) space complexity since it also takes n unit space to place all the node.

In the postlab, when forming a middleearth object, it will first take a vector of strings that has the size of n (number of cities) and a 2d array that contains the distance and n indexs for the cities which will take a total of O(2N+N^2) space complexity. In the permutaition method, it will also take a n amount vector that holds the value of the route travelled with a space complexity of O(N).

Traveling Salesman:

The algorithm we used to find the shortest path in the inlab is a brutal force method which will take a O(N!) runtime algorithm which is really time inefficient. In order to optimize the solution, there are a few exact algorithms as well as approximate algorithms that help to solve TSP. One of the exact algorithm solution is called Held–Karp algorithm which is a dynamic programming algorithm that can reduce the runtime to O(n^2*2^n). It uses recursion method to gradually seperate the entire trip into pieces and find the minimum distance for each section in order to find the ultimate minimum distance.

Based on the property “E*very subpath of a path of minimum distance is itself of minimum distance”*. It first will collect all the possible subpath to the destination beside the starting point. For example, if we want to find the min path from Bree back to itself and in that world also contains Isengard, and Minas Tirith, It will first collects all the subsets : empty set, Isengard, Minas Tirth, (Isengard, Minas Tirth) (Minas Tirth , Isengard), (Isengard, Minas Tirth, Bree). Then it will find the distance from Bree to Isengard and the distance from Bree to Minas Tirth and recurse over until it reaches to the subset that has the destination and finally compute the minimum path it takes.

One of the other method is called the Nearest neighbour algorithm, because it is an approximation to the solution, it may not always give the optimal solution to the problem .In the method it will first choose the starting vertex on the graph then finds out find the shortest edge toward any vertex that has not been visited Then we move to that vertex and repeat the same action again until all of the vertex has been visited. This way we may be able to find the shortes path. As it was called the “greedy algorithm” It may sometimes miss the shorter path since it only concerns the minimum distance in the vertex scale instead of the entire graph. However, at its best it may be able to compute at a O(N^2) runtime.

Finally, Christofides's algorithm will be able to also approximate the solution with a runtime of O(N^3). It will first create a minimum spanning tree based on the triangular ineuqality (a,b)t<t(b,c) +t(a,c). After finding the tree, we then will convert the tree into an Eulerian graph and delete the repeated route. In the end, we then able to find the minimum path