//Name:Shahroz Imtiaz

//Email ID:si6rf

//File Name: postlab11.pdf

//Date:12/6/2018

Complexity analysis

Pre-Lab:

When looking at the time complexity of my pre-lab implementation, the first thing my program does is read in the file that the user supplied, which takes O(n) as every word has to be read in. It then checks to see if the first word we're currently looking at has already been entered in before, if it hasn't, it is added to the vector of vertices, this all takes O(n) as pushing the back of a vector is O(1). My program then checks the second word read in from the same line and sees if it's been entered before and pushes it to the back of the vector if it hasn't, which again takes O(n). Combined, this part of the program takes O(n^2). My program then sorts the vertices and prints them, which takes O(n).

When looking at the space complexity of my pre-lab implementation, my vertex class takes ((8 bytes(pointer to vertices) + 4 bytes(int indegree) + SIZE_STRING bytes (which depends on how long the string is)))* NUMBER_OF_ADJ_VERTICES (this forms the Adjacent list I use). My main function takes the aforementioned * NUMBER_OF_VERTICES (which forms the vector I use to keep track of the vertices entered) + topSort, which takes NUMBER_OF_VERTICES * SIZE_OF_VERTEX class, whose size was mentioned before.

In-Lab:

When looking at the time complexity of my in-lab implementation, the first thing my program does is create the middle earth world, which takes O(1) because while the code does have to account for each city being added, the run-time for push_back() on a vector is constant. Then in O(n) a vector is created from getItinerary(), which return the vector of cites. I then make a copy of this vector which also takes O(n). I then compute the initial distance for the path which takes O(n) and the I sort my vector which takes O(n*log(n)). Next the next_permuation() method is called, which takes O(n!) as every possible combination for the min path is tried resulting in my program taking O(n!).

When looking at the space complexity of my in-lab implementation, the middle earth object created takes (4*3) bytes, which int # of cities,xsize,ysize, + (4*xsize) bytes + (4*ysize) byes + (SIZE_OF_STRING*NUM_OF_CITIES) bytes + (8*NUM_OF_CITIES) bytes, which

keeps track of distance, + (SIZE_OF_STRING + 4) * NUMBER_OF_INDICES bytes. Then the rest of my main takes (SIZE_OF_EACH_CITY_STRING * # of cites) bytes + 4 bytes (for the min path length) + 4 bytes (for current path length) which is in my computeDistance() method.

Acceleration techniques

There are a number of acceleration techniques one can use to increase the speed of their traveling salesman implementation. Three of these techniques are the Held–Karp algorithm, Branch and bound algorithm, and Nearest neighbor algorithm.

The Held–Karp algorithm is a dynamic programming algorithm, which means our Traveling salesman problem is broken down into smaller sub-problems. We then use the fact that every sub-path of a path of minimum distance is itself of minimum distance to recursively build up a minimum-path solutions, starting from the smallest subproblem. All of these sub-problems are stored in a data structure and if you ever need to compute it again for a bigger path, you don't. Instead, you look at your data structure and just pull the results you computed earlier, thus saving computation time. Though this implementation takes up a lot of space, it is ultimately faster resulting in the worse-case run-times of O(2^n n^2) and the space complexity of O(2^n n).

The Branch and bound algorithm works by recursively splitting the search space (the space used to for finding the path) into smaller spaces via branching to form a tree. The algorithm then uses higher and lower bounds, which are usually determined by some heuristic algorithm, to prune through the search space eliminating sections of the search space that yield results higher or lower than the bound. The algorithm relies on a data structure such as a queue or similar (stack) to hold partial solutions. While the queue is not empty, the algorithm removes a node off the queue and tests it's distance with the bounds, if the node yields a lower result that the results estimated using a heuristic algorithm, then this node is now the shortest path, else for each node connected to this node, check their results, if their lower bound is higher than the upper bound of the problem, discard them, since it's impossible for them to ever yield the optimal solution. And if the bound isn't higher, then add this particular node to the queue and keep repeating. If the bounds aren't carefully selected, this algorithm can result in an exhaustive search O(n!), else it has the running time of how many possible solutions it has to check by looping through them on the queue.

The last technique is different than the previous techniques because the nearest neighbor algorithm is an approximation algorithm. Though it does yield a solution, it's usually not the optimal solution. The way it works is you randomly start somewhere in the list of cities you have. You then find the nearest unvisited city to that city and then set that city to your current city and mark it visited and continue repeating this process until all cities have been visited. The running time of this algorithm is O(n) but be careful because this algorithm can sometimes not find a path even when one exists.