Justin Choi (jc8mc)
CS 2150 – 106
April 27th 2017

postlab11.pdf

In Lab 11: Graphs, we explored two different algorithms: the topological sort and traveling-salesman problem for the pre-lab and in-lab, respectively.

For the pre-lab, topological sort, when analyzing the time complexity, in the main function of topological.cpp, it read in two strings from the file, which takes Big-Theta (n), in which n is the number of strings that was in the file. As the string is read, they are pushed onto a vector, which also causes a runtime of Big-Theta (n) since the strings might not be present in the vector and it might have to be copied when the vector is full. Next, the program iterates over the vector running at Big-Theta (n) because it has to go iterate through each element in the vector. The program then reads the file again, and then it pushes the element onto a list, which both causes linear time, Big-Theta (n). After these steps, the sort method loops through a list and sorts. Then a stack is popped in a loop to print out the results. These are both linear time, Big-Theta (n).

In order to find the space complexity, in the Graph class, there are a vertices stored as 4 byte ints and a list that consists of number of vertices, which are 4 bytes each. Thus, this results in a total of 8 * the number of vertices for the Graph class. The addEdge() function contains 2 ints of 4 btyes each. The printSort() function contains a total of 5 * the number of elements. The main method creates a list of strings, which are 4 byes each, and a map of elements that are 4 bytes mapped to another element of 4 bytes size and an object of a Graph.

For the in-lab, the traveling-salesman problem, when analyzing the time complexity, we know that there are around 30 cities. Thus, we can safely assume that the vector is not full, and it does not have to be resized for this case. Thus, when we push back the name of the cities into the vector that holds cities, it has a constant running time, Big-Theta (1). After that, it has access to a 2D array, which runs based on the number of cities, since it accesses two vectors in the

position at the x-axis and the position at the y-axis, which runs at constant time, Big Theta (1). Therefore, the constructor operates at a running time of Big-Theta (n), linear time. For the next_permutation() method, since there are n! permutations in which n is the number of cities to consider when creating an itinerary of cities to travel to find the shortest path, the overall running time is Big-Theta (n!).

In order to find the space-complexity for the in-lab, it contains 4 ints of 4 bytes each for the Middle Earth code that was given (4 bytes * 4 bytes * indices). 1 vector contained every city, in which each city was 4 bytes of size (4 bytes * num_cities). And there was another vector containing 4 * xsize bytes and another vector containing 4 * ysize bytes (4bytes * xpos and 4bytes * ypos). Float distance was 4 bytes. So, the total is (4 * xpos) + (4 * ypos) + (4 * num_cities) + (16 * indicies) + 16 for the Middle Earth class. In the main function, it contained the Middle earth class, and there was a vector, dests, of cities that were 4 bytes of size each (4 bytes * num_cities), and a string of 4 bytes, and another vector of cities that were 4 bytes each (4 bytes * num_cities), and a float that hold the distance which is 8 bytes. For the computeDistance() function, it had a float of 8 bytes to hold a return value and 2 strings of 4 bytes each. So, there were a total of 16 bytes.

For the second part of the lab, we explored acceleration techniques for the traveling-salesman problem. The first one that I researched is Markov chains. This technique uses a local search algorithm to make the inputs more manageable. Sub-itineraries are created by constructing more efficient routes based on prior routes rather than creating new and random routes. Thus, it tries to create routes by local minimum distance, and this step is repeated. By adding this acceleration technique, the run time is Big-Theta (n) (http://setosa.io/ev/markov-chains/).

A second option is the nearest neighbor algorithm. This algorithm computes a path from the start to the next closest location. However, it may be very limiting and inaccurate since it looks at the graph not as a whole. By beginning at the start node and progressing to the next closest node, you repeat these steps until there are no nodes left to visit, and this is overall Big-Theta (n), linear time (https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm).

A third acceleration technique is the cutting-plane method. The cutting plane method works by linear programming to determine an optimum solution. The solution is tested with integer solutions. This is repeatedly continued until the optimal integer solution is found. I think the runtime is probably linear time, Big-Theta (n) (http://www.seas.ucla.edu/~vandenbe/236C/lectures/localization.pdf).