Maddie Stigler
mgs4ff
12/5/14
postlab11.pdf

Complexity Analysis:

Pre-Lab:

My topological program begins by reading in 2 strings at a time from the file. This takes Big Theta(n) where n is the number of strings presented in the file. After the strings are read, they are pushed onto a vector, which also has a runtime of Big Theta(n). This is because if the strings aren't already present in the vector, it might have to be copied and repopulated. Next, the program iterates over the vector running at Big Theta(n) because it has to go through each element in the vector. The program then reads from the file again and pushes the elements onto a list. Both of these also operate at Big Theta(n) runtime. After the initial setup is complete, the program calls the sort method that loops through a list and sorts. Finally, the stack is looped through and printed to the screen. Both of these operate at Big Theta(n) runtime.

When examining the space complexity of the topological program, I analyze the data structures in each part of the program. Within the graph class, there are a number of vertices stored as 4 byte ints and a list that consists of the number of vertices(4 bytes each). This results in a total of 8*(number of vertices) for the graph class. The addEdge() function contains 2 ints of 4 bytes each. The printSort() function contains a stack of elements 4 bytes each and an array of bool elements of 1 byte each. This results in a total of 5*(number of elements). The main function creates a list of strings 4 bytes each, a map of elements that are 4 bytes mapped to other elements of 4 bytes, and a graph object.

In-Lab:

The traveling.cpp program starts by initializing a Middle-Earth object. This object makes a vector of cities by pushing each city of Middle-Earth onto the vector. Because the number of cities never changes, the vector doesn't need to be resized, resulting in a constant runtime of Big Theta(1). After this, a 2d array is created that also runs at constant time. After the Middle-Earth constructor is called, the main function initiates a loop that loops through the dests vector to compute the total distance of the trip. This runs at Big theta (n) where n is the number of cities in the dests vector. Next, the next_permutation() function is called which finds every combination of city routes, resulting in n! runtime. Finally, the trip is printed to the screen. This requires a runtime of Big Theta(n) where n is the number of cities being printed.

In analyzing space complexity of the traveling program, I first look at the data structures within a Middle-Earth instance. It contains 4 ints of 4 bytes each, 1 vector containing every city (each is 4 bytes), another vector containing 4*xsize bytes, and yet another vectore containing 4*ysize bytes. Within the main function of the traveling.cpp file, there is an instance of Middle-Earth, a vector(dests) of cities that are 4 bytes each, a string of 4 bytes, another vector of cities that are 4 bytes each, and a float that hold the minimum distance which is 8 bytes. The computeDistance() function requires a float of 8 bytes to hold the return value as well as 2 strings of 4 bytes each.

Acceleration Techniques:

There are several possible acceleration techniques for enhancing the traveling program. One option is the nearest neighbor algorithm. The nearest neighbor algorithm computes a path from the start by always progressing to the next closest location. Because it doesn't look at the graph as a whole, it can sometimes be very limiting and inaccurate. However, due to its simple nature, it is very easy to implement. You begin at the start node and progress to the next closest. Afterwards, you mark the next closest as the current node and note that it has been visited. You repeat these steps until there are no nodes left and then you print the resulting string of nodes to the screen. Its worst case runtime is linear.

Another possible acceleration technique is a branch and bound algorithm. The branch and bound algorithm looks at the entire program before arriving at a solution. This makes it more reliable than the nearest neighbor algorithm but possibly less efficient. The algorithm explores all possible solutions as branches from the root. Each branch is then checked against the upper and lower bounds of the best solution and trashed if it can't produce a better solution than the previous one. It works by initializing a partial solution, looping through all the remaining solutions and replacing the greatest solution with better one. Data structures such as a stack queue and a priority queue can be used to implement it. The runtime for this is the number of possible solutions the algorithm has to loop through.

A final acceleration technique is the cutting-plane method. This is slightly more complicated than the other two but has been successful with previous traveling salesman problems. The cutting plane method works by using linear programming to determine an optimum solution. That solution is then tested for being an integer solution. If it isn't, this is considered a cut and the current non-solution is no longer an option. This continues until the optimal integer solution is found. The cutting plane method has a runtime of the number of possible linear program solutions.

Any of the three of these acceleration techniques would enhance my program. I think the branch and bound algorithm would be the most feasible because it is the closest to the brute force method already applied. This would increase the runtime of the traveling salesman program by analyzing the whole graph first. It would change the runtime from the number of possible combinations to just the number of possible branches.

Sources:

http://en.wikipedia.org/wiki/Nearest_neighbour_algorithm
http://en.wikipedia.org/wiki/Branch_and_bound
http://en.wikipedia.org/wiki/Cutting-plane_method