//Robert Alexander, RMA3MZ, postlab11

Post-Lab 11

Time Complexity

1. Pre-Lab

The first thing that the topological main file does is search the list of nodes (helper class created for this lab, holds a vector with a list of all of the adjacent nodes, as well as a name and indegree) to see if one of the strings it read in was already present, this takes O(n), where n is the number of nodes in the nodeList. It then does a pushback for the string if it isn't already present in the vector nodeList, which has a running time of O(n), since it might have to copy the whole vector over and repopulate it. Then it repeats both operations for the second string in the line. This leads to a running time of O(n^2). We also must consider the actual sorting of the node vector. In the topological algorithm, a queue is created and nodes are pushed onto it. If all nodes are pushed onto the queue, then the running time for this operation is O(n). The rest of the running time for the topolical sort is completely dependent on the makeup of the graph, since the running time of the loop depends on the amount of adjacent nodes to each node. However, this number will be constant and will not have the same impact that O(n^2) does, so the overall running time will be O(n^2).

2. In-Lab

For the in-lab we know that we have 33 cities, so we can assume that the vectors that create the middle earth world will never have to be resized. Thus when we push back the names of the cities into the vector cities, is has a constant running time of O(1). Then it computes a 2-d array, which runs based on the the number of cities, since it accesses the two vectors xpos and ypos, which run at a constant time. Overall, the constructor requires a running time of O(n).

Now we can consider the major contributor to the overall running time, the *next_permutation* call. There are n! permutations to consider, where n is the number of different cities (ignoring the minor change that not including the starting city has, n-1). This running time overshadows all other aspects of the in-lab program. Thus the running time is O(n!).

//Robert Alexander, RMA3MZ, postlab11

Space Complexity

1. Pre-Lab
    - Node Class
        - String – 4 bytes
        - Vector holding adjacent nodes – 100 elements * 4 = 400
        - Int indegree – 4 bytes
        - Overall 408 bytes for the node class
    - Main
        - Vector of node pointers – 100 * 4 bytes= 400 bytes
        - 2 bools=2 bits
        - Overall 400.2 bytes in main
    - Topological
        - Queue- probably holds max of ~ 20 nodes ~100 bytes
    - Overall=908.2 bytes

2. In-Lab
    - Middle Earth
        - 4 ints=16 bytes
        - 1 Vector=4*num_cities
        - 1 Vector=4*xsize
        - 1 Vector=4*ysize
        - Overall 16+(4*num_cities)+(4*xsize)+(4*ysize)
    - Main File
        - Middle Earth=look above
        - 1 Vector (dests)=4*num_cities
        - String first=4 bytes
        - Vector finalIt=4*num_cities
        - Float smallestDist=8 bytes
        - Float tempDist=8 bytes
        - Overall 20+2(4*num_cities)
    - Compute Distance
        - Float distance=8 bytes
        - 2 strings=8 bytes
        - Overall 16 bytes
    - Overall
        - 52+2(4*num_cities)+(4*num_cities)+(4*xsize)+(4*ysize) bytes

//Robert Alexander, RMA3MZ, postlab11

Acceleration Techniques

1. Minimum Spanning Tree

This method of acceleration makes use of a minimum spanning tree to reduce the number of vertices visited in the tour. The simplest way to think about this implementation is that a node on the tour is removed and a minimum spanning tree is constructed with the starting city as the root. Then the smallest arcs (vertices) between the remaining nodes are set as the most efficient path. Then you can reconnect the removed node by adding the smallest arcs that connect the node to the other nodes in the spanning tree. You can continually repeat this until you find the most efficient route using minimum spanning trees. There a multiple different algorithms that can be used to construct a minimum spanning tree. This method proves to be much more efficient because you are not continually revisiting the same path multiple times when attempting to determine the most efficient route.

2. Analyzing the Cities before running

A non-exact method of accelerating the speed of this program would be to make some determinations before hand about the most optimal routes to use. For example, when adding a new node, we could create a new field that holds the closest node to it. Thus when running the program, we would not have to iterate between every single permutation, but only the different permutations between the best possible paths between nodes. This would be slightly less efficient, but more exact than my next proposed acceleration technique.

3. Nearest Neighbor

This technique involves starting from the first city and finding the shortest edge connecting the start city to another city, then marking that city as visited and moving to the next city of the shortest path. Once all cities are marked as visited you now have a fairly short route created. While it is not nearly as accurate as a brute for method, or even my second method above, it still provides a fairly accurate and useful method for determining the shortest route.

Sources

- http://en.wikipedia.org/wiki/Nearest_neighbour_algorithm
- http://demonstrations.wolfram.com/TheTravelingSalesmanProblem4SpanningTreeHeuristic/
- http://www.youtube.com/watch?v=BmsC6AEbkrw