// Dion Niazi dn3gy 27 04 2017

Complexity Analysis

For my pre-lab code, topological.cpp, there were a lot of things I did to complete this algorithm. First I made a struct called courseNode, which can hold the name of the course and a pointer to the next course. Then I created a vector of courseNode pointers and then I create a while loop to read the whole file I have a stack to save all the pointers to the next course and later pop them off when I fix the specific courseNode. This was the implementation of an adjacency list. Problem was that I was unable to produce an algorithm straight from an adjacency list, so I started to implement an adjacency matrix. Once the matrix was made, I then went through a loop and checked if the whole column of an array was 0, then print that out and make that whole column ones, but its row all zeros. This worked fine for the multiple sample trials even though this seems like a weird implementation of it. That's really it of that program and it seems that this program is a bit longer than most topological sort programs. The running time in big theta seems to be big theta n^3 because my implementation was different and 2 loops within another loop. The space complexity seems to be big theta n^2 for the matrix because it's a 2D array and big theta n for the vector of courseNodes because we don't know how many courseNodes will be the vector.

For my in-lab code, traveling.cpp, there were a couple things I did for the algorithm. I first sorted the dests vector from the second element in the vector to the end. Then I made a loop to go through each permutation from the second element to the end of the dests vector. I compute the distance for each permutation and then check if it's the smaller distance. I then print out the smallest distance and then print out the paths of taken to reach the shortest path. The big theta overall seems to be big theta n! because accessing an element in the vector is big theta constant time and printing out from the whole vector is big theta n and computing the distance is big theta n!, so the highest one is n! meaning that's the overall big theta runtime. The space complexity for the vector is big theta constant space because we know how much cities we are putting into the vector but the print vector is big theta n because we don't know how many cities are in the shortest path

Acceleration Techniques

An approximation technique such as Christofides' algorithm can help speed up TSP by a huge amount and with a small amount of error. What the algorithm does is by using graph theory it helps improve How this algorithm works is by one method is by finding the minimum spanning tree, then create duplicates for every edge to create an Eulerian graph, then find an Eulerian tour for the graph, and lastly convert to a TSP. This helps speed up the problem by at most 1.5 times the optimal. Meaning an estimation to my code that it would be 1.5 times faster. There wasn't no big theta running time describe, but rather big O and its n^3, so maybe it is big theta n^3 as well.

An acceleration technique could be to use the branch and bound algorithm. The world record for the TSP was solved using the branch and bound algorithm, so it must be good. It seems that the running time would still be exponential so big theta 2^n. The time this algorithm may take on my program is by maybe by 2 times, I'm estimating. The algorithm works by recursively splitting the search space into smaller spaces, then minimize the function on those spaces and keep track of the bounds on the minimum that it is trying to find and eliminate solutions that it can prove will not contain the optimal solution.

Another approximation technique for TSP is by using the nearest neighbor algorithm, which is a greedy algorithm. It quickly yields an effective short route and it is done by starting on an arbitrary vertex and find the shortest edge with the current vertex and an unvisited vertex, V. Set the current vertex to V and mark V as visited. If all the vertices in the domain are visited, then terminate the program. Repeat from finding the shortest edge. The big theta run time is big theta log V, where v is the vertices. This algorithm, if implemented in my program may make it log time faster.