Group Members: Barbara Hazlett, Wing Yee Leung, Ethan Spiro
CS 325:Project 4 - Traveling Salesman
Language: C++
Compiling/running Instructions: g++ -std=c++0x cs325_project4_tsp.cpp -o run
./run <textfile.txt>

Objective

Our goal was to design and implement a method for finding the best tour that we could in the limited time frame. We know that it is very difficult to find a method that is both efficient and accurate, so before we began we decided to set a few minimum requirements that our algorithm should meet. We aimed to achieve in a worse-case scenario a tour no longer than 40% of the shortest distance. Efficiency-wise, we aimed to have an algorithm better than O($n!$) (brute-force/exponential method.)

Process

We decided to start exploring algorithms that were not exact algorithms. We decided our best bet was to choose an approximation algorithm, which are known to quickly yield good (but approximate) results. We decided that the nearest neighbor algorithm was one we were confident we could implement within the given time frame, and one that would yield solutions that fits to our goals in efficiency and accuracy.

Once we had that implemented, we realized that by choosing a random city as a starting point that this algorithm produced solutions quickly. From our resources we knew that on average the result produced is 25% more than the shortest distance. Even though it was better than our worst-case scenario, we wanted to do better than that.

We decided we could spare some efficiency to improve on our approximate solution. If we iterate through all the cities, using a different city as a starting point for our nearest neighbor algorithm and select the best solution, we will improve on our approximation. We will lose on efficiency, but that it is still far better than brute force. The nearest neighbor algorithm iterates through all of the cities twice so it has a big-Oh of O(n^2). We call it once for each city so the big-Oh of our algorithm is O(n^3).

How it Works

We declared a struct (called cityType) which would hold all the necessary information for a city - id, x coordinate, y coordinate, and a boolean to determine if said node had been visited. We also declared a vector to hold all of these cities and vectors (of ints) for the tour and shortest tour. We then call readInfo to populate our vector with cities.

When we have read everything from the file and have completely populated the map vector we call our algorithm. First we clone our map and call this new vector not_visited. In addition, we create a vector of integers (to hold ID's) to represent the tour. We set the boolean value of all cities in not_visited to false, and then take the node passed in and treat it as the start. We take the start node, set it to true, push its id onto our tour and then begin searching for the next closest node that has not been visited (closest node calculated by rounded integer euclidean distance in the calcDistance function). When the closest next node is determined, we mark it as true, add it to the tour, and treat it as the current node. We add the distance to our distance counter and then repeat this process. This search process is done in a loop that continues to execute until there are no cities marked as not having been visited. When we reach the end of the loop we add the final distance between the node passed in (start node) and the node we finished on in order to “return home”. If this tour has yielded a new shortest tour, we save it.

After we have run this algorithm with every node as a starting point, we output a file with our shortest tour, writing the total tour distance on the first line and the city order on the subsequent lines.

Results

We ran all of the test examples that were supplied with the project document to verify our program worked correctly and so that we could compare our results to the best tour lengths. Our results were 15% to 22% higher than the best tour lengths which was about what we expected.

Reflections and Possible Improvements

We understand that we can improve on both the efficiency and accuracy of our algorithm. We realized that our algorithm was quite slow when we had a large number of cities (approx n > 10000.) Our original plan was to implement 2 optimizations, one that would have better accuracy and one with better efficiency. We would time these algorithms and plot time and accuracy information on a graph. Once we know at what point these algorithms became too inefficient for us to find a solution within x amount of time, we could say “for n < x cities, we will run algorithm 1, for n > y cities, we will run algorithm 2.” We would choose the slower but more accurate algorithm when we had a small number of cities, and the less accurate but more efficient one when we have a greater number of cities.

We were interested in exploring an implementation of the ant colony optimization that models behavior in real ants to find short paths between food sources and their nests. However, we were unsure if we were capable of implementing this optimization before the due date. In addition there are such specific probability functions for an ant to choose a path depending on pheromones deposited, evaporation route, amount of ants etc… that we did not feel like we had an original solution - so we dropped that idea.

Other options we considered were the brute force algorithm, for accuracy, for cases when cities < 10. We also considered the pairwise exchange (or 2-opt technique) and cheapest link algorithm for better efficiency.