Dennis Lang

Spring 2020

CIS 7

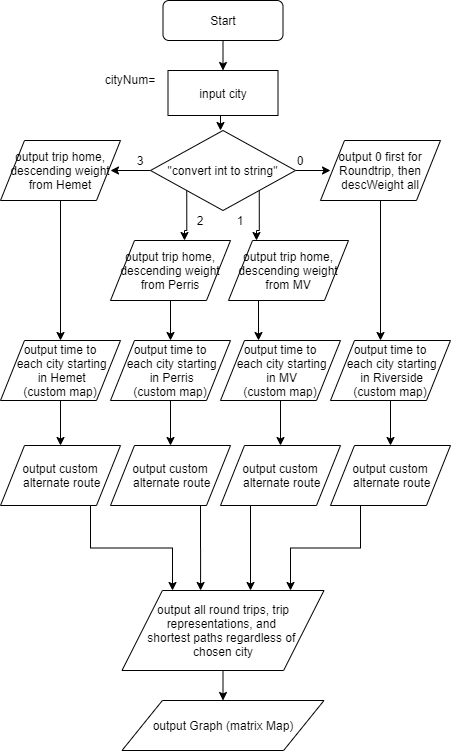
Team Dennis

Final Project

This program seeks to plan out the work schedule of a solar panel salesman in the Inland Empire. Given that the salesman’s home is located in Riverside and his targeted cities are Hemet, Moreno Valley, and Perris, the objective is to find out all of the different variations of trips the salesman can make and what is the best route to take. Different routes include the shortest path, breadth-first path, and lowest cost path. Using matrices and adjacencies in the form of graphs in C++, the program will offer the user different options and use the stored information about the cities (average distance in between cities in miles and minutes). This program could feasibly be used for any type of traveling salesman or company that is involved with shipping logistics. The ability for a commerce company to detail their route down the minute with utmost efficiency is coveted and is an entire industry known as logistics. Every minute and mile saved by a company that processes millions of dollars in transactions in day adds up, which is greatly lucrative for a company. This program seeks to solve this issue for a single person, but it could easily be adapted for any employee of the company as well as for providing valuable data for the shipping department.

The major data structure implemented into this project is the graph, along with the use of vertices and edges. Using an undirected graph, weights can be applied to the edges of the lines between vertices, allowing for different traversal strategies and a breadth-first sorting algorithm. An adjacency list is also used to "draw" the graph into the program, giving it an idea of what vertices are connected and which are not. The weights of the edges are also stored in this list. The program uses traversal as well to determine the routes that are possible and giving a possible route for the solar salesman. The program also uses decision trees internally, as the shortest route and breadth-first route are both calculated by the weighted edges which determines what the program tells the user, depending on the given input. An array of city names is used to keep the cities linked to their individual numbers. Multiple iterators are used to sort through the city names and key out the cities either sequentially using incident vertices. An implementation of Dijkstra’s algorithm is the main player in the means of finding the paths based on the edge values.

This program is quite limited as of this time unfortunately. The program only contains information about the four relevant cities in Riverside County, but other locations can theoretically be used in the future in the form of an implemented Google or Apple Maps API. Additional cities could be implemented into the current array based on the distance away from the home city, or perhaps using a database with a primary key and stored distances. Another limitation is the amount of options available and the use of the console. Choosing options currently is only available by typing into the console. A true GUI would be desirable in such a program, and people would likely prefer to create custom routes and maps. Other options could include random routes and more precise mapping by neighborhood, as well as storing of location data and connections to the sales database, providing a true logistics service for a budding company or delivery service. Different users could also login in the future, and the program could also allow saved paths and different assigned paths for each salesman.



Pseudocode:

//Assign all variables (distance between cities in minutes and/or miles)

//optimally use this weighted edge for all calculations

edges[] = [0,1,20],[0,2,14],[0,3,39]..

//create functions

Printout city.weight, city.value, city.next,

//Define smallest edge, breadth-first, cheap trip

//weight the edges with travel time in minutes on average

//prompt user for key in for the current city

cin cityNumber;

Cin cityNumber;

//Breadth first: create a list; mark cities traveled to with a variable, pop non-visited cities in stack

If(city[I+10] == notVisited && connectedViaList); //travel to next closest city

Cout >> city.closest >> city.close >> city.far (using loops);

//Shortest route: create a graph, set distance to a integer/variable, look for values in the list and search //for the smallest weight value to travel to next

If(city[I].val >> city[I+1].val);

Cout >> city.current >> city.closest >> city.nextclosest >> city.nextclosest >> city.home;

//Cheapest trip: same as above but consider route with multiple variables by adding the combined

//weight values as possible checksums against the individual edges

//If(city[1].val + city[2].value >> city[3].val) && (city[I].val >> city[I+1].val);

If city[1] -> city[2] >> city [1]+ city[3] -> city[2], cout city[1]- > city[2].weight;

//If Riverside is chosen as first city be sure to remove it from the top of the stack and/or make sure it is //the last city visited too

Cout >> Riverside >>city.next >> city.next >> city.next;

//Dijkstra example pops the smallest value

For(I=0,I<N, I++)

Cout << city[I].smallestWeight <<endl;

Pop city[I];

I++;

//else do nothing, as it always makes Riverside last otherwise

Else >> cout >> city.current >>city.next;

//print out map with weights

For j=0,j<N,j++ ;

Print map;

Map[] = weightedList/weightedlist[0];

Void mapMaker;

Cout vertex1, vertex2, weight[j];

j++;

Return 0;