**Albert Nguyen**

**CIS 7**

**12/16/2020**

**Team Me**

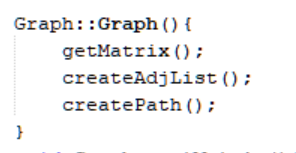
**Inland Empire Solar Sales Travel**

The final project that was chosen for Team Me is the Inland Empire Solar Sales Travel. The problem is to have a salesman start at their home node, Riverside, and travel to the locations of Perris, Moreno Valley, and Hemet. Afterwards, they need to return to their home node. The problem being solved is to find what would the shortest path be and what the cheapest path would be.

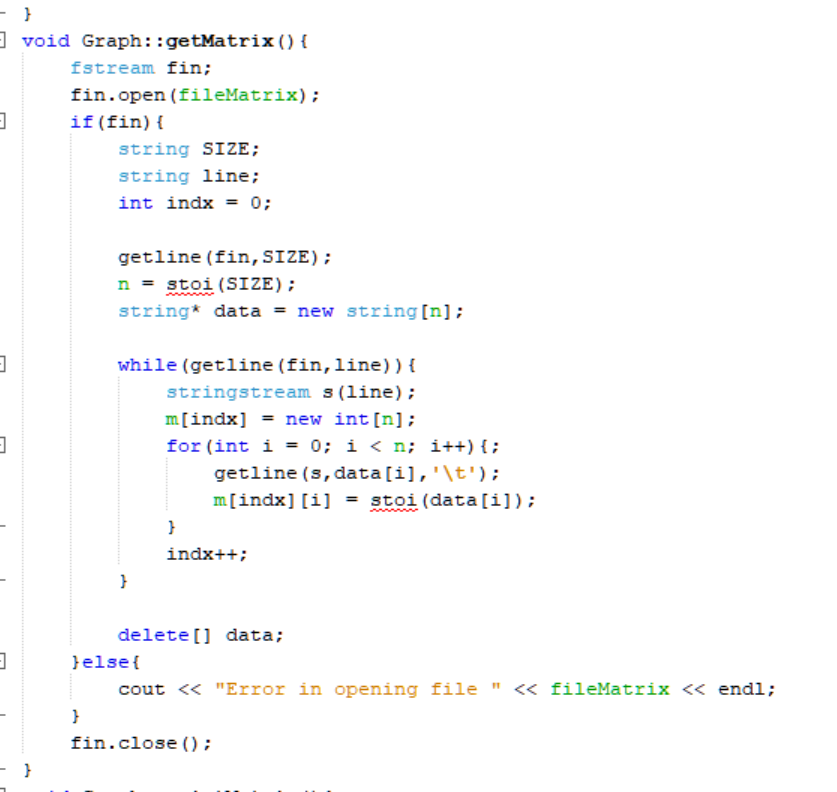
**Github Link**

<https://github.com/albuut/TSP>

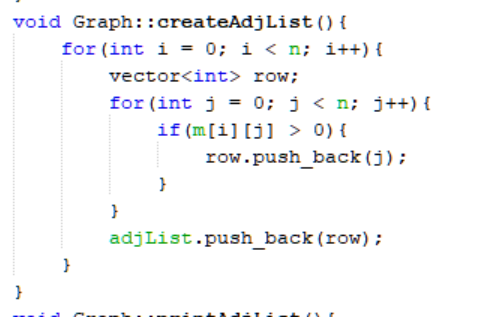
**Code/Solution Explanation**



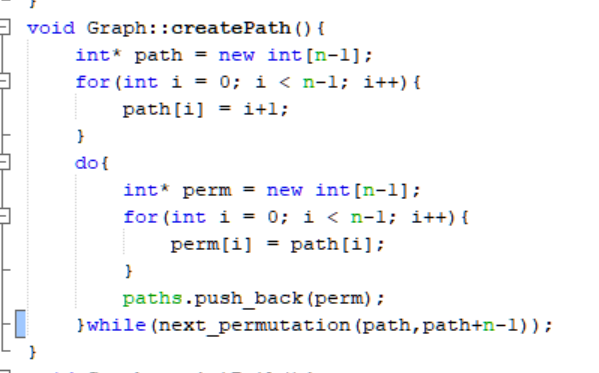
I created a graph class which calls on three functions when constructed.



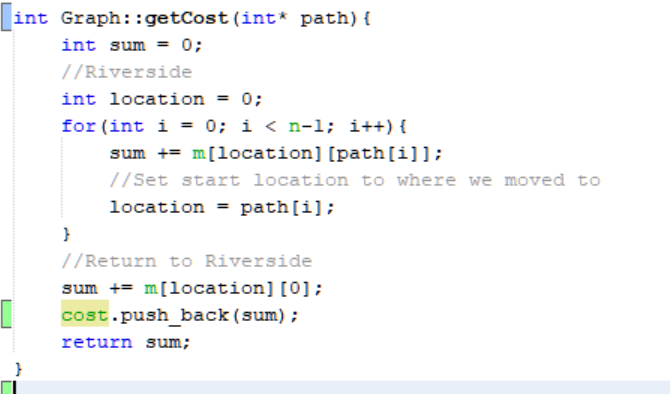
I stored my matrix in a .txt file that’s inside of the project. This is where if I was to change values or etcetera it’d be done there. The first line contains the size of my N x N matrix and the following lines contain the rows of my matrix. I have the function read until the end of the program and I grab each line. Every time I read a line, I’ll put it into a stringstream and have it delimited based on tabs so that It’d be able to grab the individual values inside of it. All the values are then stored in the 2D array m.



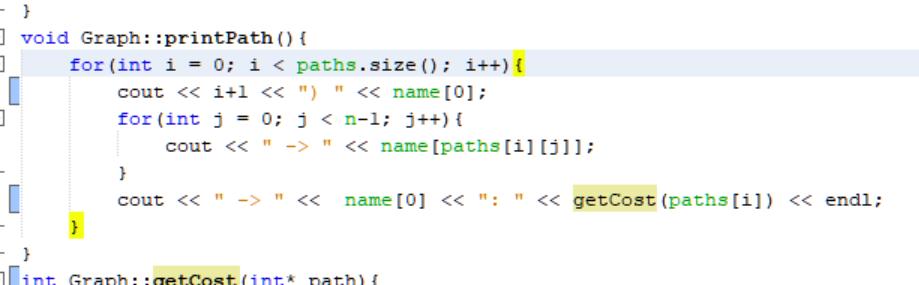
Create adjacency list is used to sort the matrix to check what values are non->zero for each row. If the value is greater than 0. It will push the value of the index into the vector adjList. So, each row in the adjList contains what values that the node is connected to. For this problem, it has the property in which each node is connected to all the other nodes. This property is important due to the problem requires you to reach every node



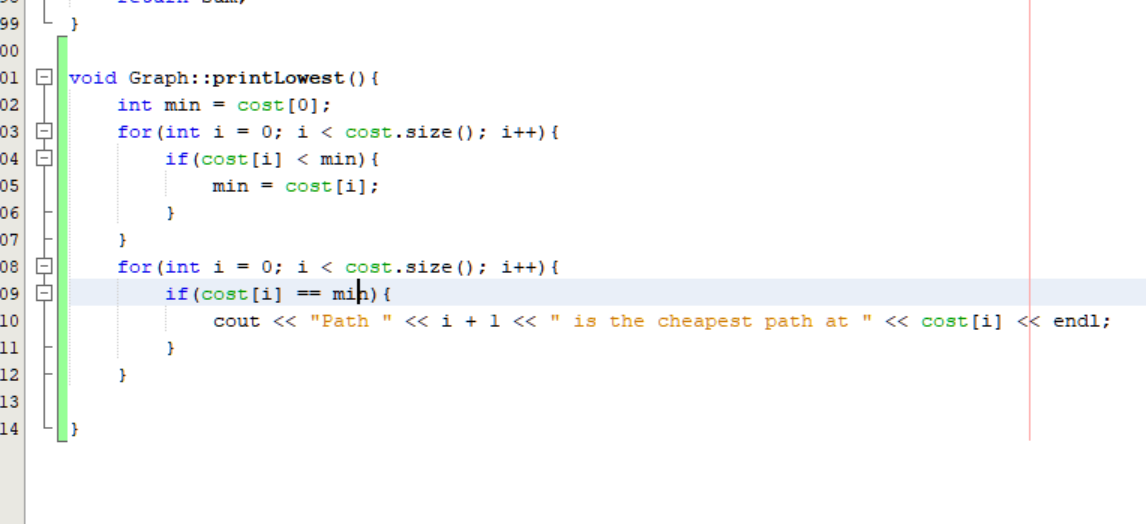
Observing the property that this problem links to all other node beside itself and that we can only visit each node once. We know that we must start at Riverside and end at Riverside. This means that the variations of paths could be treated as permutation problem due to the connectivity between the notes. This is done by creating an array that contains values of the index of the three locations. We then use the next\_permutation stl and push each permutation into a paths vector. We can predict that the number of permutations available would be 3! Which is what we found when running the program.



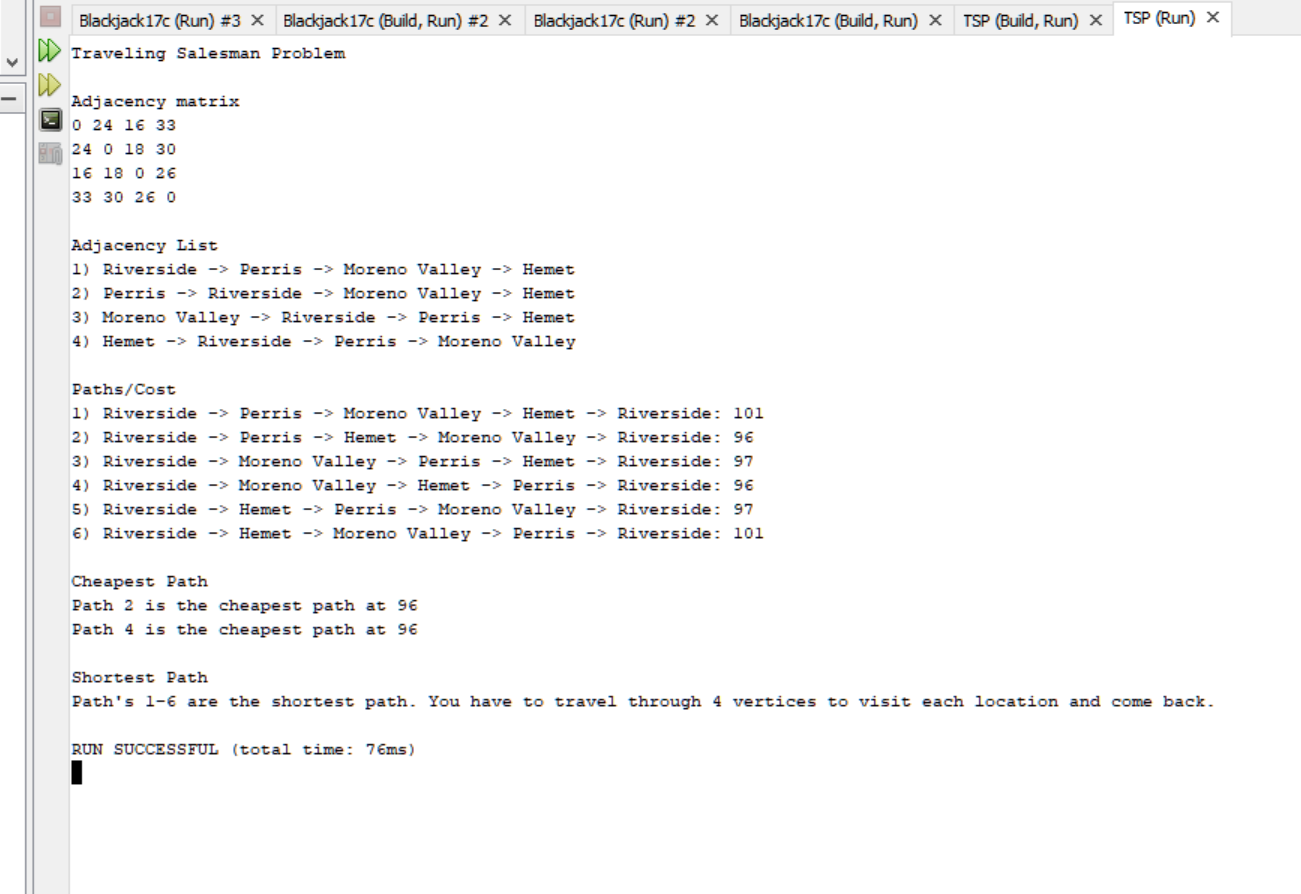
To compute the cost, it takes to travel each of these paths, I created a getCost function. This takes an array called path. This path consists of integer values of each location that’s contained in M. I initialize a variable sum in order to hold the summations of the cost and initialize location at 0. The main reason why I choose to initialize location at 0 is because I have it ordered in a manner that Riverside is where you start. So at index 0, it’d give you the row which contain the weights of Riverside to other nodes. I think iterate through my paths. Location would indicate where the salesman is currently located. It accesses the row for weights and access the next location, which is contained in path[i]. This returns the weight of going from where you are now, to somewhere else. Afterwards, it updates location to be the location you moved to and it starts the cycle again, eventually summing the total cost it takes to transverse this path. Every time it computes a path, it passes the cost into a cost vector.



I then just iterate through all my paths by printing the values. Every time a path is posted, it also calls the getCost function and passes the pass through to return the sum of the cost. This allows the user to be notified of the value of the path cost.



My printLowest() function finds the minimum value in all my cost and prints them out. This is done by initializing min as the first value for minimum and iterating through and comparing. Doing it this way also allows for multiple paths if they’re the same value to be posted.



This is the program’s output. It first prints out the adjacency matrix with all the weighted graphs. It then prints out the adjacency list in which we notice that all the locations are connected to the others. Afterwards it posts all the possible permutations of paths and their cost. It then outputs the cheapest and shortest paths. Due to the property of the graphs being bidirectional, we can extrapolate that if the path was to travel through the nodes in the same order but reversed. The values should be the same. Therefore we have two path’s which are considered the cheapest path and it appears to be duplicates of cost for the other paths.

**Programs Objective**

The program’s objective is to use the adjacency matrix that is stored in a text file to generate an adjacency matrix which is then converted into an adjacency list and used to brute force the paths to generate the cost of each of the paths. After doing all the calculations, it is then displayed for the user to see. This kind of problem doesn’t require any additional input from the user. The purpose of the program is just to show the user what path is the lowest cost to take if they were.

**Discrete Structure**

The discrete structure that’s added is the graph. It’s currently being stored as an adjacency matrix. The graph is used to extract data to know how far each location is relative to each other and if there’s a path/edge that connects them to allow them to travel from one location to the other. This also uses permutations in order to brute force and final all the possible paths rather than manually testing all of them and checking.

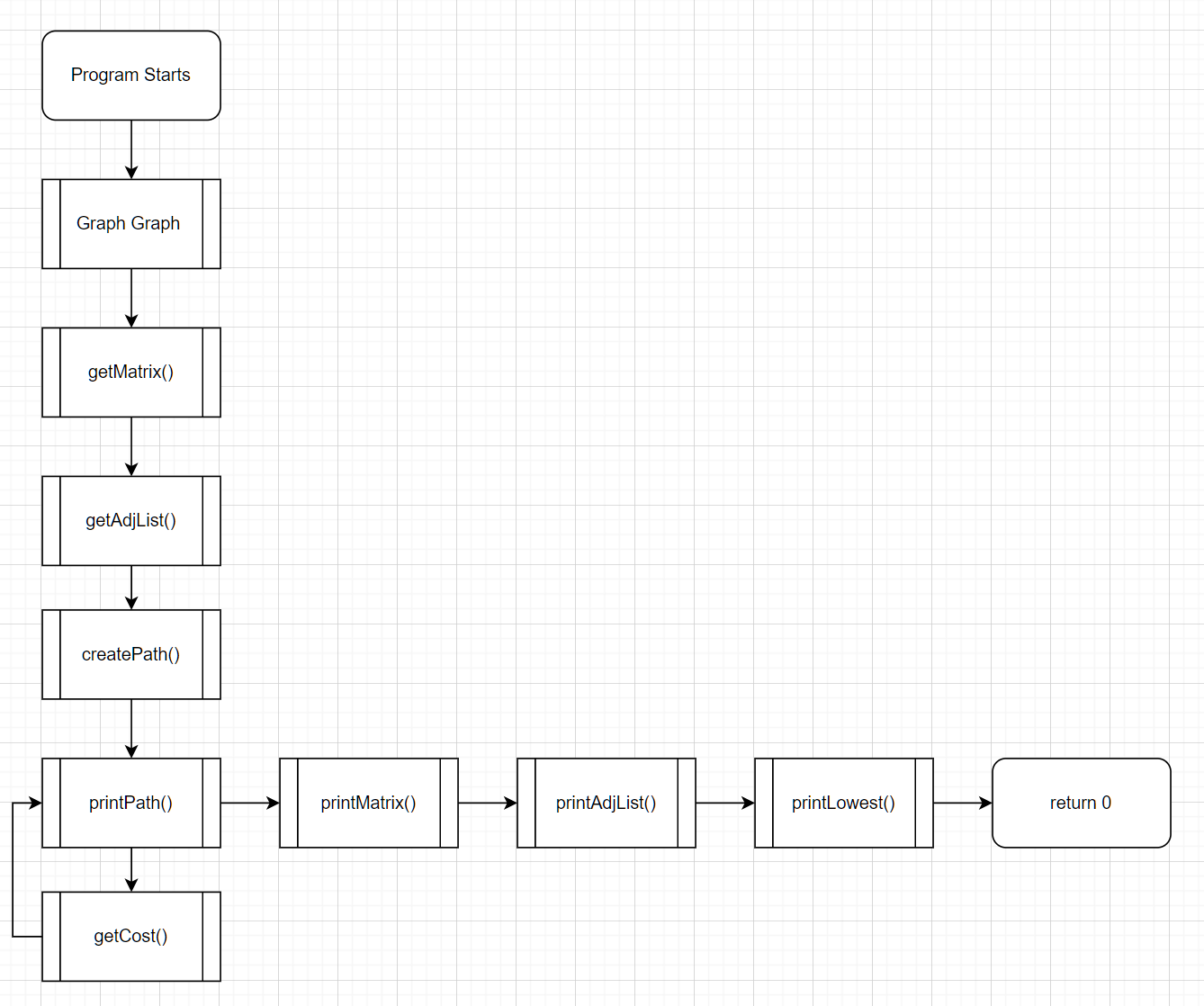
**Limitations**

The limitations of this program is in the constraint, for example if we had a problem in which we didn’t have a complete path and it didn’t allow backtracking when there’s only one node connecting. That’d leave a massive issue. Also, the usage of permutations becomes much more complicated and no longer easy if not all of the values are connected to all other nodes. An example of this would be if terrain would not allow the salesman to go directly to another city but must access the city to get to another. The code would not be able to run in such a naïve way. The code also uses nested for loops which would result in n^2 time complexity. This is a real bad running time since the more cities we had the longer it is going to take to do the computation.

**Fixes**

The backtracking issue would not be fixable, it’s a constraint of the problem itself since we’re only allowed to visit each city once. However, the permutation issue would be fixable. Rather than calculating the possible combinations all at once. We’d split off the values in n-trees and save whichever values that’d fill criteria of visiting all of the locations, have connecting edges, and have a path that returns them to the original location. Running time is also a problem that’s not easily fixed since each new permutation would scale it exponentially. In order for us to reduce the time, we probably would have to give up accuracy for an approximation using approximation/heuristic algorithms which are not quite in the scope of the class, this would probably work in the same manner as merge sort where we divide and conquer starting at different nodes at once but this would require certain conditions to be met for this problem to work such as being able to travel freely between all cities.

**Flow Chart**

****

**Psuedo Code**

Call an object of Graph

Graph Initialize a variable for size

Graph initializes values for filename to be read

Initalize a 2d vector

Initialize vector of arrays for paths

Initialize vector for cost

Initialize names

Initialize a 2d array for adjacency list

In constructor

Read and generate a matrix

Read Into file

Delimit the file to get values

Save into a matrix

Generate adjacency list

Read into previous saved matrix

Create list of paths that the files can take

Read off the matrix

Save the connections in matrix

Permutate possible paths

Notice that all the paths are connected to all vertices

So only have to permutate the three that are not the first

First will always be riverside, and end will always be riverside

Compute cost

Take permutated path list and pass it through a loop to grab cost values and sum them

Each pass needs to save the previous location to access the new row in which you are computing

Sum/Save and return these values to be printed and to be analyzed

Sort your cost array to find the minimal value

Print out Matrix/Adjacency List/Paths/Which are the cheapest paths/Which are the shortest paths

Return 0;