Write-up for Group C Assignment

Roll numbers : 2959,2960,2961,2969

Name of the project :

CABLE NETWORK DESIGNING USING GRAPH

1. Problem Statement:

Designing a television cable network and finding optimum costs of leasing out cables between various houses.

2.Details about designing:

- **CLASSES:**

We have three classes in our program:

1.class line:

Each instance of this class consists of 2 houses and the cost of laying out a cable between them.

2.class connection:

This class helps in creating the actual cable 'network' between various houses by accepting data like no. of houses and no. of maximum cables available. This class includes functions to find shortest distances/ costs of cabling between various houses.

3.public class project:

This class contains the main function of the program and asks the user for various menu options to choose the desired area over houses which are to be included in the connection.

- **FUNCTIONS:**

Following are the functions (class-wise) used in our program. :-

A.Functions in class line :

A1.line(data):

This is actually a parameterized constructuctor which accepts house nos. To and from which a connection is possible and the leasing cost of cable between them.

** *Time complexity* : O(1)

A2.void display():

This function simply displays an accepted value ,i.e. a line data.

** *Time complexity* :O(1)

A3.void displayPath():

This is another display function that displays a line data but a different format.

** *Time complexity* :O(1)

B.Functions in class connection :

B1.void createNeighbourhood():

This function accepts the total no. of houses to be included in the connection and costs of cabling between each pair of houses. The entries are sorted in increasing order of their costs and stored in a list. For each entry, it adds to an adjacency matrix and an adjacency list and thus, a graph is created.

* * *Time complexity* :O(n^2) {where n:total no. of houses}

B2.void displayPotentialLines():

This is simply a display function which displays all the data accepted in the function above.

* *Time complexity* :O(1)

B3.void addNewHouseToConnection():

This function can be used to add a new house to the existing network using the same logic used in the first (create) function.

** *Time complexity* :O(1)

B4.void findMinCableRoueKruskals():

This function is actually more suitable for a sparse network. It creates a minimum spanning tree , that is a connection including all the houses with minimum costs of cables used. The tree that we are making usually remains disconnected.

** *Time complexity* :O (E log V) {E :no. of cables , V:no. Of houses}

B5.void display_adj():

This function displays the adjacency matrix of the data accepted to be used in the prim's algorithm (in function B7).

** *Time complexity* :O(n^2) {where n:total no. of houses}

B6.int minKey():

This function ,for a given vertex, returns its adjacent vertex which has the least cost of cabling and which has not been visited yet.

* *Time complexity* :O(n) {where n:total no. of houses}

B7.boolean createMST():

This function returns if a vertex is already present in the MST being constucted or not.

** *Time complexity* :O(1)

B8.void printMST(int parent):

This function makes use of prim's algorithm to find MST ,i.e a network consisting of all houses and uses cables such that the total cost of leasing is minimum. This algorithm is more suitable for denser graphs as compared to the algorithm in function B4. The tree that we create remains connected. A random vertex is taken as input from the user and the next cheapest vertex is added subsequently to the tree.

This function constructs and prints the network created from the above function. Also ,the minimum total cost possible is displayed.

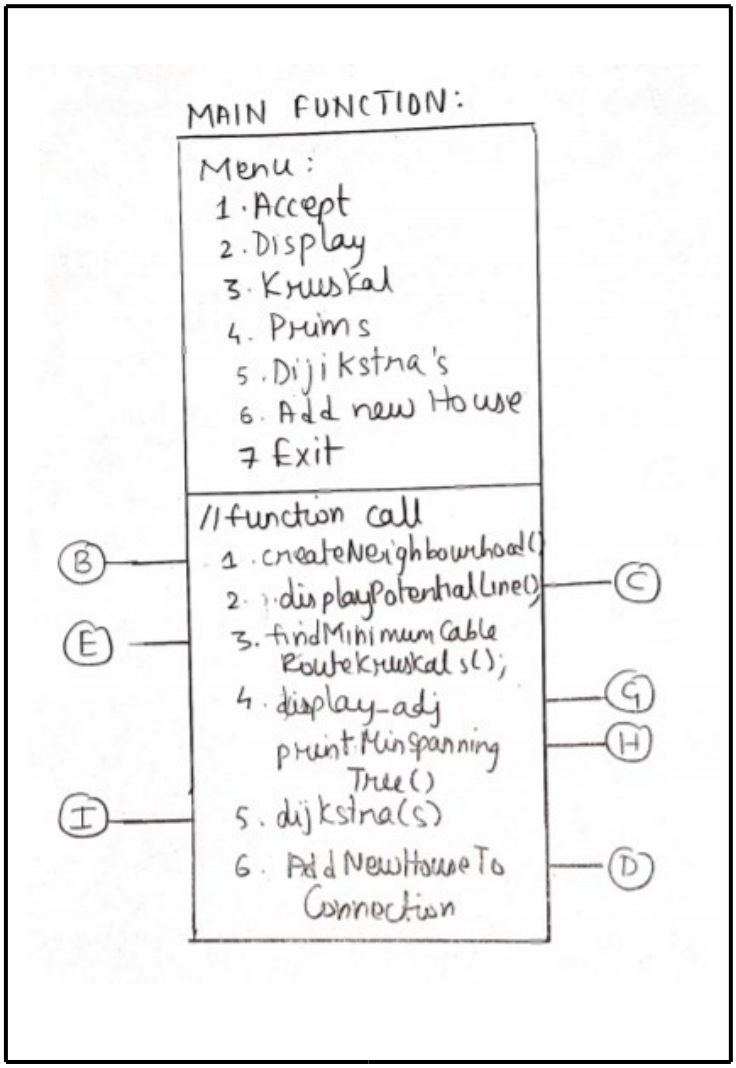
** *Time complexity* :O(n^2) {where n:total no. of houses}

B9.void dijikstas():

This function is used to find the minimum cabling cost of any destination house from a given source house taken from the user.

- * *Time complexity* :O(n^2) {where n:total no. of houses}

3.BLOCK DIAGRAM



FUNCTIONS:

void createNeighbourhood()
void displayPotentialLine()
void addNewHouseToConnect()
void findMiniumCableRoute KowsKals()
void display_adj()
int minKey(int kKey[], Boolean mstSet[])
boolean createMST(int u, int boolean inMST[])
void printMinSpanningTree()
void dijkstra(int sourceVerte
void printDijkstra(int sourceVertex, int[] kKey)

void displayPath()

Display cost from particular house number to particular house number

Void display()

(A) Display distance from particular house number to particular house number

Void createNeighbourhood()

(B)
- accept the cost per meter
- accept no. of houses
- accept no. of possible paths
- loop (untill the no. of possible paths)
  - accept from house number
  - accept to house number
  - accept distance from source to dest.

- store those values in list as well as in array
- sort the list using collections.sort from collection frameworks.

Void displayPotentialLines()

(C)
- loop(until the no. of possible path)
- call to the display function in line class

Void addNewHouseToConnection()

(D)
- accept the no. of new houses
- accept no of possible paths from new house
- loop(until no. of possible paths)
  - update matrix and list.
- sort the list

Void findMinimumCablePathKruskals();

(E)
1. Sort all the edges in non-decreasing order of their weight
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it
3. Repeat #2 until there are (V-1) edges in the spanning tree

int minKey(int key[], Boolean mstSet[])

(F)
- loop(until no. of houses)
  - if (mstSet(v) = false && key[v] < min)
    {
      min = key(v)
      min_index = v;
    }
- return minIndex;

Void display-adj()

(G)
- Displays the adjacency matrix

Void printMinSpanningTree()

(H)
1. Initialize the minimum spanning tree with a vertex chosen at random
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree.
3. Keep repeating step 2 until we get a minimum spanning tree

Void dijKstra(int s)

(I)
1. Accept src & dest node
2. loop(next not visited to dest)
   1. mindist = infinity
   2. find the next node which is not visited and has min distance from source
   3. Mark visited of next
   4. Update distance of every other node by considering dist(next) as new distance

4. SHORT NOTE ON DATA STRUCTURE USED :

We have used graph data structure for our project . A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. "*A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes."*

Graphs are used to solve real-life problems that involve representation of the problem space as a network. Examples of networks include telephone networks, circuit networks, social networks (like LinkedIn, Facebook etc.). For example, a single user in Facebook can be represented as a node (vertex) while their connection with others can be represented as an edge between nodes. Each node can be a structure that contains information like user's id, name, gender, salary, etc.

Graphs can be commonly represented using two ways :

1. Adjacency matrix : An adjacency matrix is 2D array of V x V vertices. Each row and column represent a vertex.
2. Adjacency list : An adjacency list represents a graph as an array of linked list.

The most common graph operations are:

1.Check if element is present in graph
2.Graph Traversal
3.Add elements(vertex, edges) to graph
4.Finding path from one vertex to another

5. OPERATIONS PERFORMED :

5.1 Logic related to operations :

A. Kruskal's Algorithm –

1.Sort all edges in the graph G in the order of their increasing weights
2.Repeat V-1 times //as MST contains v-1 edges
{
a.Select the next edge with minimum weight from the graph G
b.If(no cycle is formed by adding the edge in mST i.e the edge connects the different connected components in MST
c.Add the edge in MST
}

B. Prim's Algorithm –

1. Create mst[] to keep track of vertices included in MST.
2. Create key[] to keep track of key value for each vertex. Which vertex will be included next into MST will be decided based on the key value.
3. Initialize key for all vertices as MAX_VAL except the first vertex for which key will 0. (Start from first vertex).
4. While(all the vertices are not in MST).
    4.1 Get the vertex with the minimum key. Say its vertex ***u***.
    4.2 Include this vertex in MST and mark in mst[***u***] = true.

4.3 Iterate through all the adjacent vertices of above vertex ***u*** and update the keys if adjacent vertex is not already part of mst[].

5. We will use Result object to store the result of each vertex. Result object will store 2 information's.

5.1 First the parent vertex, means from which vertex you can visit this vertex. Example if for vertex **v**, you have included edge **u-v** in mst[] then vertex **u** will be the parent vertex.

5.2 Second weight of edge u-v. If you add all these weights for all the vertices in mst[] then you will get Minimum spanning tree weight.

A. Djikstra's Algorithm –

1. Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3.While *sptSet* doesn't include all vertices

3.1 Pick a vertex u which is not there in *sptSet* and has minimum distance value.

3.2 Include u to *sptSet*.

3.3Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.