Project 2

(Processing a Weighted Undirected Graph)

Algorithms and Data Structures

(ITCS 6114 – 091)

Fall 2012

By

Abdul Aaquib

#800795370

11/27/2012

**Problem Statement**

Write a program to process a weighted undirected graph as follows:

1. Read in the number of vertices V and the number of edges E of the graph followed by its E edges, each in the form u, v, w where 1 <= u, v <= V & w > 0 representing an edge uv with weight w.

2. Set up and print the adjacency matrix representation of the Graph.

3. Determine whether the graph is connected.

4. Find a minimum spanning tree for each component and print the minimum spanning forest in adjacency matrix representation (regardless it has just one or more than one components).

5. If the graph is connected, use Priority-First Search to find a shortest path tree from vertex 1. Print the adjacency matrix representation of the tree.

You should document your program, analyze the complexity of your algorithms, and show the outputs from sample data sets that Mr. Zhou (TA) will provide you next week.

**Solution**

**1. Input:**

The program provides the user with 2 options to input the graph data:

1. Read the data from a Text file
2. Manually input the data

To read the data from the text file, the file must be located in C:\

E.g. if the name of the file is ‘graph1-Proj2.txt’, then path of the file will be “C:\graph-Proj2.txt”

Also, while entering the filename, the user should enter the full name of the file (with type)

E.g. Enter the filename: graph1-Proj2.txt

The format of the data in the text file should be as follows:

* First line should be the number of vertices
* Second line should be the number of edges
* A list of edges starting from the third line. Each edge should be on a new line and in the following format:

u,v,w

where u and v are vertices (representing edge uv) and w is the weight of the edge. u,v and w should be separated by commas.

If the user selects the second option, he should enter the data manually in the console as prompted.

**2. Adjacency Matrix:**

The program reads the input data and constructs an adjacency matrix representing the graph. The adjacency matrix is stored in a 2-dimensional array and printed to the screen using the method printAdjacencyMatrix().

**3. Is the Graph connected?**

The program uses Depth-First-Search to determine whether the graph is connected or disconnected.

**Depth-First Search**

In depth-first search, we start the graph traversal at an arbitrary vertex and go down a particular branch as deep as possible. When we reach a dead end, we back up and continue along another branch.  In this way we visit all vertices, and all edges. However, if a graph is disconnected, DFS won't visit all the vertices. Hence, we can use this algorithm to find out whether the graph is connected by counting the number of vertices visited by the algorithm. The DFS algorithm is as follows:

void dfsearch(int u,int &count)

{

color[u]=GRAY; //vertex is in progress

count++; //increment count for each vertex

for(int j=1;j<=vertexCount;j++)

{

if(adjacencyMatrix[u][j]!=inf)

{

if(color[j]==WHITE) //if the vertex has not yet been processed

dfsearch(j,count);

}

}

color[u] = BLACK; //dfs has finished processing the vertex

}

bool isConnected()

{

int count=0;

dfsearch(root,count);

if(count==vertexCount)

return true;

else

return false;

}

dfsearch() function is called once with root as the starting vertex. If the graph is connected, then the algorithm will visit all the vertices. If the graph is disconnected, it does not visit all vertices.

The DFS algorithm traverses the graph and increments ‘count’ for every vertex processed. Hence, after DFS finishes executing

* If count == no. of vertices, the graph is connected
* If count <= no. of vertices, the graph is disconnected

**Complexity of Depth-First Search used above**

In the above DFS algorithm, dfsearch() is a recursive function and is called **n** times for each of the **n** vertices in the graph, where **n** is the number of vertices. As we have recursion and we try visiting all the vertices on each step, the worst-case time is **O(n2).**

By using an adjacency matrix we need n2 space for a graph with **n** vertices. We also use an additional array to mark visited vertices, which requires additional space of **n**. Thus the space complexity is **O(n2).**

**4. Minimum Spanning Tree**

In a weighted undirected graph, a spanning tree is a collection of edges that connects all the vertices in the graph. A minimum spanning tree is a spanning tree that has the lowest total edge-weight of all possible spanning trees. A minimum spanning forest of a graph is the graph consisting of the minimum spanning trees of its components. There is at least one minimum spanning tree for all connected and weighted graphs.

This program uses Kruskal’s algorithm to find the minimum spanning tree of the input graph.

**Kruskal’s Algorithm**

Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected, weighted, undirected graph. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component).

The algorithm starts by initializing a set of |V| trees. During the process of building the final spanning tree we keep a forest with |V| trees, where each tree has a single vertex. It finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u, v) of least weight. A safe edge is an edge which connects two separate trees in the forest. Kruskal’s algorithm is a greedy algorithm, because at each step it adds to the forest an edge of least possible weight.

int Find\_Set(int x, int parent[])

{

if(x != parent[x])

parent[x] = Find\_Set(parent[x], parent);

return parent[x];

}

//Kruskal's Algorithm to find the Minimum Spanning Tree of the Graph.

void kruskal()

{

int u, v;

totalWeight = 0;

for (int v = 1; v <= vertexCount; v++) //Initialization

{

parent[v] = v; //Create a new set whose only member is pointed to by v

}

//Sort the edges into non-decreasing order by weight

sort(edgeList.begin(), edgeList.end());

for(int i=0; i<edgeCount; i++) //for each edge

{

u = Find\_Set(edgeList[i].second.first, parent);

v = Find\_Set(edgeList[i].second.second, parent);

if(u != v) //if u and v belong to different trees

{

MSTedgeList.push\_back(edgeList[i]); //Add edge to tree

parent[u] = parent[v]; // Union (combine the two trees)

//Construct Adjacency Matrix for MST

MSTMatrix[edgeList[i].second.first][edgeList[i].second.second] = edgeList[i].first; MSTMatrix[edgeList[i].second.second][edgeList[i].second.first] = edgeList[i].first;

totalWeight += edgeList[i].first; // increment total weight

}

}

}

The algorithm needs to sort the edges and uses a disjoint set data structure (parent[]) to keep track which vertex is in which component. Find\_Set() function returns the set to which the vertex belongs. The above code takes an edge and finds the sets to which the 2 vertices of the edge belong. If the sets u and v are not the same (u != v), then it adds the edge to the MST and combines the 2 sets (parent[u] = parent[v]).

When the algorithm finishes executing, MSTedgeList will contain all the edges of the minimum spanning tree and MSTMatrix will contain the Minimum Spanning Tree in adjacency matrix representation.

**Complexity of Kruskal’s Algorithm used above**

Initializing a set (parent[]) in the first for loop takes O(1)time. We need O(|V|) operations to build the initial forest with |V| trees each containing one vertex.

Next, the sorting of edges is done in O(|E| lg |E|) time.

The second for loop is executed at most |E| times.

Then for each edge, two Find\_Set () operations are done to determine the sets (or trees) to which the two end-points of the given edge belong. Since the set contains at most |V| elements, the running time for the Find\_Set() operations is O(lg |V|) in the worst case.

If the the two end-points are in different trees (u != v), we add the edge to MSTedgeList, combine the two trees, insert the edge into MSTMatrix and increment the totalWeight. All these operations can be done in O(1) time.

Therefore, the second for loop takes O(|E| lg |V|) time.

Hence the total running time of the algorithm is **O(|V|) + O(|E| lg |E|) + O(|E| lg |V|).**

Disregarding the lower term O(|V|), we get O(|E| lg |E| + |E| lg |V|).

In the worst case, |E| = O(|V|2). Hence, lg |E| = O(lg |V|2) = O(2 lg |V|) = O(lg |V|).

Thus, the total running time for Kruskal's algorithm in the program is: **O(|E| lg |V|).**

**5. Shortest Path Tree**

The shortest path tree of a connected, undirected graph G is a spanning tree of G such that the distance from root v to any other vertex u in the tree is the shortest path distance from v to u in G. We can use a Priority-First Search algorithm like Dijkstra’s or Bellman-Ford algorithm to find the shortest path tree of a graph.

This program uses Dijkstra’s algorithm to generate the shortest path tree of the input graph.

**Dijkstra’s Algorithm**

Dijkstra’s algorithm solves the single-source shortest-paths problem on a weighted, directed or undirected graph *G* = *(V, E)* for the case in which all edge weights are nonnegative. It calculates the shortest distances from the root to all other vertices in the graph producing a shortest path tree. It is optimal i.e. it will find the single shortest path from the root to any vertex. Also, it is an uninformed search, meaning it does not need to know the target node beforehand.

For each vertex v, Dijkstra's algorithm keeps track of three pieces of information:

visited[v]: An array that indicates that the shortest path to vertex v is known. Initially, visited[v]=false, for all v Є V.

distance[v]: distance[v] is the length of the shortest known path from the root to v. When the algorithm begins, no shortest paths are known. During the course of the algorithm, candidate paths are examined and the distance array is modified. Initially, distance[v] = ∞ for all v Є V and distance[root] = 0.

parent[v]: The predecessor of vertex v on the shortest path from root to v. Initially, parent[v] is unknown for all v Є V.

void Dijkstras()

{

int i,u,w;

root = 1; //root=1

for(i=1;i<=vertexCount;i++) //Initialize

{

visited[i]=0;

distance[i]=adjacencyMatrix[root][i];

parent[i] = NULL;

}

distance[root]=0;

i=1;

while(i<=vertexCount)

{

u=Extract\_Min();

visited[u]=1;

i++;

for(w=1;w<=vertexCount;w++)

{

if(((distance[u]+adjacencyMatrix[u][w])<=distance[w]) && !visited[w]) //Relax

{

distance[w]=distance[u]+adjacencyMatrix[u][w];

parent[w] = u;

}

}

}

//Construct Shortest Path Tree

for(int i=2;i<=vertexCount;i++)

{

shortestPathTree[i][parent[i]] = adjacencyMatrix[i][parent[i]]; shortestPathTree[parent[i]][i] = adjacencyMatrix[parent[i]][i];

}

}

//Returns the closest vertex

int Extract\_Min()

{

int w=1,j;

double min=inf;

for(j=1;j<=vertexCount;j++)

{

if(distance[j]<min && !visited[j])

{

min=distance[j];

w=j;

}

}

return w;

}

The Dijkstra's algorithm above proceeds in phases. The following steps are performed in each pass:

1. From the set of vertices with visited[v] = false, select the vertex *v* having the smallest distance[v].
2. Set visited[v] = true.
3. For each vertex *w* adjacent to *v* for which visited[w] != true, test whether distance[w] is greater than distance[v] + adjacencyMatrix[v][w]. If it is, then set distance[w] = distance[v] + adjacencyMatrix[v][w] and set parent[w] = v.

The algorithm terminates after |V| passes are completed at which time all the shortest paths are known.

**Complexity of Dijkstra’s Algorithm used above**

In the first for loop, we initialize visited[], distance[] and parent[] for each vertex in |V|. Hence, the first for loop takes O(|V|) time.

The while loop is executed at most |V| times.

The u=Extract\_Min() operation takes O(|V|) time (since we have to search the entire distance[] array for the closest vertex).

The next step updates the distance[] and parent[] arrays for all vertices adjacent to u. Since, we check for all the vertices, this step takes O(|V|) time.

Lastly we construct the Shortest Path Tree using the parent[] array which takes O(|V|-1) time or O|V| time.

Hence, the total running time of the above algorithm is O (|V|) + O ( |V| \*( |V| + |V| + |V| ) ) which is equal to O(|V|\*|V|) = **O(|V|2).**

**Output**

**Input 1: (graph1-Proj2.txt)**

20 (no. of vertices)

25 (no. of edges)

19,1,3 (list of edges u,v,w where u,v is the edge and w is the weight)

1,20,5

1,2,7

2,4,7

4,5,10

17,5,5

18,5,20

8,3,3

7,8,2

16,7,6

7,10,5

4,10,7

6,11,6

11,12,10

9,13,12

7,13,10

13,14,8

10,14,50

14,11,100

15,11,12

6,4,5

1,9,20

8,4,15

17,12,33

15,18,5

**Output 1:**

How do you want to input the data?

1. Read data from text file.(the file should be located in C:\ e.g. "C:\graph1-Proj2.txt")

2. Manually enter the data

Enter choice (1/2): 1

Enter the name of the file: graph1-Proj2.txt

The Adjacency Matrix Representation of the Graph is as follows:

inf 7 inf inf inf inf inf inf 20 inf inf inf inf inf inf inf inf inf 3 5

7 inf inf 7 inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf

inf inf inf inf inf inf inf 3 inf inf inf inf inf inf inf inf inf inf inf inf

inf 7 inf inf 10 5 inf 15 inf 7 inf inf inf inf inf inf inf inf inf inf

inf inf inf 10 inf inf inf inf inf inf inf inf inf inf inf inf 5 20 inf inf

inf inf inf 5 inf inf inf inf inf inf 6 inf inf inf inf inf inf inf inf inf

inf inf inf inf inf inf inf 2 inf 5 inf inf 10 inf inf 6 inf inf inf inf

inf inf 3 15 inf inf 2 inf inf inf inf inf inf inf inf inf inf inf inf inf

20 inf inf inf inf inf inf inf inf inf inf inf 12 inf inf inf inf inf inf inf

inf inf inf 7 inf inf 5 inf inf inf inf inf inf 50 inf inf inf inf inf inf

inf inf inf inf inf 6 inf inf inf inf inf 10 inf 100 12 inf inf inf inf inf

inf inf inf inf inf inf inf inf inf inf 10 inf inf inf inf inf 33 inf inf inf

inf inf inf inf inf inf 10 inf 12 inf inf inf inf 8 inf inf inf inf inf inf

inf inf inf inf inf inf inf inf inf 50 100 inf 8 inf inf inf inf inf inf inf

inf inf inf inf inf inf inf inf inf inf 12 inf inf inf inf inf inf 5 inf inf

inf inf inf inf inf inf 6 inf inf inf inf inf inf inf inf inf inf inf inf inf

inf inf inf inf 5 inf inf inf inf inf inf 33 inf inf inf inf inf inf inf inf

inf inf inf inf 20 inf inf inf inf inf inf inf inf inf 5 inf inf inf inf inf

3 inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf

5 inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf

Generating the Minimum Spanning Tree using Kruskals Algorithm...

The order of the selected edges are as follows

( 7,8 ): 2

( 8,3 ): 3

( 19,1 ): 3

( 1,20 ): 5

( 6,4 ): 5

( 7,10 ): 5

( 15,18 ): 5

( 17,5 ): 5

( 6,11 ): 6

( 16,7 ): 6

( 1,2 ): 7

( 2,4 ): 7

( 4,10 ): 7

( 13,14 ): 8

( 4,5 ): 10

( 7,13 ): 10

( 11,12 ): 10

( 9,13 ): 12

( 15,11 ): 12

Minimum Weight: 128

The Minimum Spanning Tree in adjacency matrix representation is as follows

inf 7 inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf 3 5

7 inf inf 7 inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf

inf inf inf inf inf inf inf 3 inf inf inf inf inf inf inf inf inf inf inf inf

inf 7 inf inf 10 5 inf inf inf 7 inf inf inf inf inf inf inf inf inf inf

inf inf inf 10 inf inf inf inf inf inf inf inf inf inf inf inf 5 inf inf inf

inf inf inf 5 inf inf inf inf inf inf 6 inf inf inf inf inf inf inf inf inf

inf inf inf inf inf inf inf 2 inf 5 inf inf 10 inf inf 6 inf inf inf inf

inf inf 3 inf inf inf 2 inf inf inf inf inf inf inf inf inf inf inf inf inf

inf inf inf inf inf inf inf inf inf inf inf inf 12 inf inf inf inf inf inf inf

inf inf inf 7 inf inf 5 inf inf inf inf inf inf inf inf inf inf inf inf inf

inf inf inf inf inf 6 inf inf inf inf inf 10 inf inf 12 inf inf inf inf inf

inf inf inf inf inf inf inf inf inf inf 10 inf inf inf inf inf inf inf inf inf

inf inf inf inf inf inf 10 inf 12 inf inf inf inf 8 inf inf inf inf inf inf

inf inf inf inf inf inf inf inf inf inf inf inf 8 inf inf inf inf inf inf inf

inf inf inf inf inf inf inf inf inf inf 12 inf inf inf inf inf inf 5 inf inf

inf inf inf inf inf inf 6 inf inf inf inf inf inf inf inf inf inf inf inf inf

inf inf inf inf 5 inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf

inf inf inf inf inf inf inf inf inf inf inf inf inf inf 5 inf inf inf inf inf

3 inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf

5 inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf

Graph is a Connected Graph

Calculating the shortest path tree from vertex 1(source) using Dijkstra's Algorithm (Priority First Search)...

The shortest distances from the source(vertex 1) to all other vertices are as follows

(1)--(2): 7

(1)--(3): 31

(1)--(4): 14

(1)--(5): 24

(1)--(6): 19

(1)--(7): 26

(1)--(8): 28

(1)--(9): 20

(1)--(10): 21

(1)--(11): 25

(1)--(12): 35

(1)--(13): 32

(1)--(14): 40

(1)--(15): 37

(1)--(16): 32

(1)--(17): 29

(1)--(18): 42

(1)--(19): 3

(1)--(20): 5

The Shortest Path Tree in adjacency matrix representation is as follows

inf 7 inf inf inf inf inf inf 20 inf inf inf inf inf inf inf inf inf 3 5

7 inf inf 7 inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf

inf inf inf inf inf inf inf 3 inf inf inf inf inf inf inf inf inf inf inf inf

inf 7 inf inf 10 5 inf inf inf 7 inf inf inf inf inf inf inf inf inf inf

inf inf inf 10 inf inf inf inf inf inf inf inf inf inf inf inf 5 inf inf inf

inf inf inf 5 inf inf inf inf inf inf 6 inf inf inf inf inf inf inf inf inf

inf inf inf inf inf inf inf 2 inf 5 inf inf inf inf inf 6 inf inf inf inf

inf inf 3 inf inf inf 2 inf inf inf inf inf inf inf inf inf inf inf inf inf

20 inf inf inf inf inf inf inf inf inf inf inf 12 inf inf inf inf inf inf inf

inf inf inf 7 inf inf 5 inf inf inf inf inf inf inf inf inf inf inf inf inf

inf inf inf inf inf 6 inf inf inf inf inf 10 inf inf 12 inf inf inf inf inf

inf inf inf inf inf inf inf inf inf inf 10 inf inf inf inf inf inf inf inf inf

inf inf inf inf inf inf inf inf 12 inf inf inf inf 8 inf inf inf inf inf inf

inf inf inf inf inf inf inf inf inf inf inf inf 8 inf inf inf inf inf inf inf

inf inf inf inf inf inf inf inf inf inf 12 inf inf inf inf inf inf 5 inf inf

inf inf inf inf inf inf 6 inf inf inf inf inf inf inf inf inf inf inf inf inf

inf inf inf inf 5 inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf

inf inf inf inf inf inf inf inf inf inf inf inf inf inf 5 inf inf inf inf inf

3 inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf

5 inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf

**Input 2: (graph2-Proj2.txt)**

10

12

1,9,3

1,2,1.2

2,5,0.5

2,3,0.8

3,6,3.1

3,10,1.5

4,9,3.2

4,5,1.5

5,7,2

5,8,5.1

10,8,8.8

6,7,5.5

**Output 2:**

How do you want to input the data?

1. Read data from text file.(the file should be located in C:\ e.g. "C:\graph1-Proj2.txt"

2. Manually enter the data

Enter choice (1/2): 1

Enter the name of the file: graph2-Proj2.txt

The Adjacency Matrix Representation of the Graph is as follows:

inf 1.2 inf inf inf inf inf inf 3 inf

1.2 inf 0.8 inf 0.5 inf inf inf inf inf

inf 0.8 inf inf inf 3.1 inf inf inf 1.5

inf inf inf inf 1.5 inf inf inf 3.2 inf

inf 0.5 inf 1.5 inf inf 2 5.1 inf inf

inf inf 3.1 inf inf inf 5.5 inf inf inf

inf inf inf inf 2 5.5 inf inf inf inf

inf inf inf inf 5.1 inf inf inf inf 8.8

3 inf inf 3.2 inf inf inf inf inf inf

inf inf 1.5 inf inf inf inf 8.8 inf inf

Generating the Minimum Spanning Tree using Kruskals Algorithm...

The order of the selected edges are as follows

( 2,5 ): 0.5

( 2,3 ): 0.8

( 1,2 ): 1.2

( 3,10 ): 1.5

( 4,5 ): 1.5

( 5,7 ): 2

( 1,9 ): 3

( 3,6 ): 3.1

( 5,8 ): 5.1

Minimum Weight: 18.7

The Minimum Spanning Tree in adjacency matrix representation is as follows

inf 1.2 inf inf inf inf inf inf 3 inf

1.2 inf 0.8 inf 0.5 inf inf inf inf inf

inf 0.8 inf inf inf 3.1 inf inf inf 1.5

inf inf inf inf 1.5 inf inf inf inf inf

inf 0.5 inf 1.5 inf inf 2 5.1 inf inf

inf inf 3.1 inf inf inf inf inf inf inf

inf inf inf inf 2 inf inf inf inf inf

inf inf inf inf 5.1 inf inf inf inf inf

3 inf inf inf inf inf inf inf inf inf

inf inf 1.5 inf inf inf inf inf inf inf

Graph is a Connected Graph

Calculating the shortest path tree from vertex 1(source) using Dijkstra's Algorithm (Priority First Search)...

The shortest distances from the source(vertex 1) to all other vertices are as follows

(1)--(2): 1.2

(1)--(3): 2

(1)--(4): 3.2

(1)--(5): 1.7

(1)--(6): 5.1

(1)--(7): 3.7

(1)--(8): 6.8

(1)--(9): 3

(1)--(10): 3.5

The Shortest Path Tree in adjacency matrix representation is as follows

inf 1.2 inf inf inf inf inf inf 3 inf

1.2 inf 0.8 inf 0.5 inf inf inf inf inf

inf 0.8 inf inf inf 3.1 inf inf inf 1.5

inf inf inf inf 1.5 inf inf inf inf inf

inf 0.5 inf 1.5 inf inf 2 5.1 inf inf

inf inf 3.1 inf inf inf inf inf inf inf

inf inf inf inf 2 inf inf inf inf inf

inf inf inf inf 5.1 inf inf inf inf inf

3 inf inf inf inf inf inf inf inf inf

inf inf 1.5 inf inf inf inf inf inf inf

**Input 3: (graph3-Proj2.txt)**

10

13

1,4,2.3

1,9,1.5

1,5,2.4

7,4,8.3

5,4,3.1

9,5,5.6

7,9,0.8

8,6,3.1

8,2,8.2

2,3,1.5

2,10,6.3

3,6,3.2

3,10,5.6

**Output 3:**

How do you want to input the data?

1. Read data from text file.(the file should be located in C:\ e.g. "C:\graph1-Proj2.txt"

2. Manually enter the data

Enter choice (1/2): 1

Enter the name of the file: graph3-Proj2.txt

The Adjacency Matrix Representation of the Graph is as follows:

inf inf inf 2.3 2.4 inf inf inf 1.5 inf

inf inf 1.5 inf inf inf inf 8.2 inf 6.3

inf 1.5 inf inf inf 3.2 inf inf inf 5.6

2.3 inf inf inf 3.1 inf 8.3 inf inf inf

2.4 inf inf 3.1 inf inf inf inf 5.6 inf

inf inf 3.2 inf inf inf inf 3.1 inf inf

inf inf inf 8.3 inf inf inf inf 0.8 inf

inf 8.2 inf inf inf 3.1 inf inf inf inf

1.5 inf inf inf 5.6 inf 0.8 inf inf inf

inf 6.3 5.6 inf inf inf inf inf inf inf

Generating the Minimum Spanning Tree using Kruskals Algorithm...

The order of the selected edges are as follows

( 7,9 ): 0.8

( 1,9 ): 1.5

( 2,3 ): 1.5

( 1,4 ): 2.3

( 1,5 ): 2.4

( 8,6 ): 3.1

( 3,6 ): 3.2

( 3,10 ): 5.6

Minimum Weight: 20.4

The Minimum Spanning Tree in adjacency matrix representation is as follows

inf inf inf 2.3 2.4 inf inf inf 1.5 inf

inf inf 1.5 inf inf inf inf inf inf inf

inf 1.5 inf inf inf 3.2 inf inf inf 5.6

2.3 inf inf inf inf inf inf inf inf inf

2.4 inf inf inf inf inf inf inf inf inf

inf inf 3.2 inf inf inf inf 3.1 inf inf

inf inf inf inf inf inf inf inf 0.8 inf

inf inf inf inf inf 3.1 inf inf inf inf

1.5 inf inf inf inf inf 0.8 inf inf inf

inf inf 5.6 inf inf inf inf inf inf inf

Graph is Not Connected.

Hence it is not possible to find the shortest path tree

**Input 4: (graph4-Proj2.txt)**

15

20

1,3,1.2

1,2,3.1

2,3,2.5

6,7,0.8

6,9,1.2

6,15,9.8

7,9,0.8

7,15,1.1

7,12,3

12,9,2.5

15,12,3.1

4,5,1.2

4,8,3

5,13,1.6

13,8,6.1

11,8,3.2

11,10,1.2

10,8,5.1

10,14,2.1

13,14,3.1

**Output 4:**

How do you want to input the data?

1. Read data from text file.(the file should be located in C:\ e.g. "C:\graph1-Proj2.txt"

2. Manually enter the data

Enter choice (1/2): 1

Enter the name of the file: graph4-Proj2.txt

The Adjacency Matrix Representation of the Graph is as follows:

inf 3.1 1.2 inf inf inf inf inf inf inf inf inf inf inf inf

3.1 inf 2.5 inf inf inf inf inf inf inf inf inf inf inf inf

1.2 2.5 inf inf inf inf inf inf inf inf inf inf inf inf inf

inf inf inf inf 1.2 inf inf 3 inf inf inf inf inf inf inf

inf inf inf 1.2 inf inf inf inf inf inf inf inf 1.6 inf inf

inf inf inf inf inf inf 0.8 inf 1.2 inf inf inf inf inf 9.8

inf inf inf inf inf 0.8 inf inf 0.8 inf inf 3 inf inf 1.1

inf inf inf 3 inf inf inf inf inf 5.1 3.2 inf 6.1 inf inf

inf inf inf inf inf 1.2 0.8 inf inf inf inf 2.5 inf inf inf

inf inf inf inf inf inf inf 5.1 inf inf 1.2 inf inf 2.1 inf

inf inf inf inf inf inf inf 3.2 inf 1.2 inf inf inf inf inf

inf inf inf inf inf inf 3 inf 2.5 inf inf inf inf inf 3.1

inf inf inf inf 1.6 inf inf 6.1 inf inf inf inf inf 3.1 inf

inf inf inf inf inf inf inf inf inf 2.1 inf inf 3.1 inf inf

inf inf inf inf inf 9.8 1.1 inf inf inf inf 3.1 inf inf inf

Generating the Minimum Spanning Tree using Kruskals Algorithm...

The order of the selected edges are as follows

( 6,7 ): 0.8

( 7,9 ): 0.8

( 7,15 ): 1.1

( 1,3 ): 1.2

( 4,5 ): 1.2

( 11,10 ): 1.2

( 5,13 ): 1.6

( 10,14 ): 2.1

( 2,3 ): 2.5

( 12,9 ): 2.5

( 4,8 ): 3

( 13,14 ): 3.1

Minimum Weight: 21.1

The Minimum Spanning Tree in adjacency matrix representation is as follows

inf inf 1.2 inf inf inf inf inf inf inf inf inf inf inf inf

inf inf 2.5 inf inf inf inf inf inf inf inf inf inf inf inf

1.2 2.5 inf inf inf inf inf inf inf inf inf inf inf inf inf

inf inf inf inf 1.2 inf inf 3 inf inf inf inf inf inf inf

inf inf inf 1.2 inf inf inf inf inf inf inf inf 1.6 inf inf

inf inf inf inf inf inf 0.8 inf inf inf inf inf inf inf inf

inf inf inf inf inf 0.8 inf inf 0.8 inf inf inf inf inf 1.1

inf inf inf 3 inf inf inf inf inf inf inf inf inf inf inf

inf inf inf inf inf inf 0.8 inf inf inf inf 2.5 inf inf inf

inf inf inf inf inf inf inf inf inf inf 1.2 inf inf 2.1 inf

inf inf inf inf inf inf inf inf inf 1.2 inf inf inf inf inf

inf inf inf inf inf inf inf inf 2.5 inf inf inf inf inf inf

inf inf inf inf 1.6 inf inf inf inf inf inf inf inf 3.1 inf

inf inf inf inf inf inf inf inf inf 2.1 inf inf 3.1 inf inf

inf inf inf inf inf inf 1.1 inf inf inf inf inf inf inf inf

Graph is Not Connected.

Hence it is not possible to find the shortest path tree

**Program**

#include <fstream>

#include <sstream>

#include <iostream>

#include <string>

#include <vector>

#include <algorithm>

#define inf 100001

#define WHITE 0

#define GRAY 1

#define BLACK 2

#define edge pair< int, int >

#define MAX 1001

using namespace std;

class Graph {

private:

double\*\* adjacencyMatrix; //Stores the graph in adjacency matrix form

double\*\* MSTMatrix; //Stores the Minimum Spanning Tree in adjacency matrix form

double\*\* shortestPathTree; //Stores the Shortest Path Tree in adjacency matrix form

int vertexCount, edgeCount;

int \*color;

// List of edges in ( w (u, v) ) format

vector< pair< double, edge > > edgeList, MSTedgeList; //Stores the List of Edges

int \*parent; //Stores the parent of each vertex

double totalWeight; //Stores the sum of the weights of all edges in the MST

int root; //root is initialized to 1 in the constructor

double distance[MAX]; //Stores the distance of each vertex from the root

int visited[MAX]; //Stores whether a vertex has been visited or not

public:

Graph(int vertexCount) //Constructor, initializes all the variables

{

this->vertexCount = vertexCount;

color = new int[vertexCount];

edgeCount = 0;

totalWeight = 0;

parent = new int[MAX]();

root = 1;

adjacencyMatrix = new double\*[vertexCount+1];

MSTMatrix = new double\*[vertexCount+1];

shortestPathTree = new double\*[vertexCount+1];

for (int i = 1; i <= vertexCount; i++)

{

adjacencyMatrix[i] = new double[vertexCount+1];

MSTMatrix[i] = new double[vertexCount+1];

shortestPathTree[i] = new double[vertexCount+1];

for (int j = 1; j <= vertexCount; j++)

{

adjacencyMatrix[i][j] = inf;

MSTMatrix[i][j] = inf;

shortestPathTree[i][j] = inf;

color[j] = WHITE;

}

}

}

//Adds an edge from vertex i to vertex j with weight w to the graph

void addEdge(int i, int j, double weight)

{

if (i > 0 && i <= vertexCount && j > 0 && j <= vertexCount)

{

adjacencyMatrix[i][j] = weight;

adjacencyMatrix[j][i] = weight;

edgeList.push\_back(pair< double, edge >(weight, edge(i, j)));

this->edgeCount++;

}

}

//Prints the Adjacency Matrix of the Graph

void printAdjacencyMatrix()

{

for (int i = 1; i < vertexCount+1; i++)

{

for (int j = 1; j < vertexCount+1; j++)

{

if(adjacencyMatrix[i][j] == inf)

cout<<"inf\t";

else

cout<<adjacencyMatrix[i][j]<<" \t";

}

cout<<"\n";

}

}

//Prints the Adjacency Matrix of the Minimum spanning tree

void printMSTMatrix()

{

for (int i = 1; i < vertexCount+1; i++)

{

for (int j = 1; j < vertexCount+1; j++)

{

if(MSTMatrix[i][j] == inf)

cout<<"inf\t";

else

cout<<MSTMatrix[i][j]<<" \t";

}

cout<<"\n";

}

}

//Use Depth First Search to count the number of vertices in the graph to determine whether the graph is connected

void dfsearch(int u,int &count)

{

color[u]=GRAY;

count++;

for(int j=1;j<=vertexCount;j++)

{

if(adjacencyMatrix[u][j]!=inf)

{

if(color[j]==WHITE)

dfsearch(j,count);

}

}

color[u] = BLACK;

}

//function to check whether the graph is connected or not

bool isConnected()

{

int count=0;

dfsearch(root,count);

if(count==vertexCount)

return true;

else

return false;

}

//Find the subtree to which a vertex belongs

int Find\_Set(int x, int parent[])

{

if(x != parent[x])

parent[x] = Find\_Set(parent[x], parent);

return parent[x];

}

//Kruskal's Algorithm to find the Minimum Spanning Tree of the Graph.

void kruskal()

{

int u, v;

totalWeight = 0;

for (int v = 1; v <= vertexCount; v++)

{

parent[v] = v; //Create a new set whose only member is pointed to by v

}

//Sort the edges into non-decreasing order by weight

sort(edgeList.begin(), edgeList.end());

for(int i=0; i<edgeCount; i++)

{

u = Find\_Set(edgeList[i].second.first, parent);

v = Find\_Set(edgeList[i].second.second, parent);

if(u != v)

{

MSTedgeList.push\_back(edgeList[i]); //Add edge to tree

parent[u] = parent[v]; // Union

//Construct Adjacency Matrix for MST

MSTMatrix[edgeList[i].second.first][edgeList[i].second.second] = edgeList[i].first; MSTMatrix[edgeList[i].second.second][edgeList[i].second.first] = edgeList[i].first;

totalWeight += edgeList[i].first; // increment total weight

}

}

}

//Prints the order of the selected edges and the minimum weight

void printMSTOrder()

{

int i, sz;

sz = MSTedgeList.size();

for(i=0; i<sz; i++)

{

cout<<"( "<< MSTedgeList[i].second.first;

cout<<","<< MSTedgeList[i].second.second<<" )";

cout<<": "<< MSTedgeList[i].first<<"\n";

}

cout<<"\nMinimum Weight: "<< totalWeight<<endl;

}

//Use Dijkstra's Algorithm to find the Shortest Path Tree with root = 1

void Dijkstras()

{

int i,u,w;

root = 1;

for(i=1;i<=vertexCount;i++) //Initialize

{

visited[i]=0;

distance[i]=adjacencyMatrix[root][i];

parent[i] = NULL;

}

distance[root]=0;

i=1;

while(i<=vertexCount)

{

u=Extract\_Min();

visited[u]=1;

i++;

for(w=1;w<=vertexCount;w++) //Relax

{

if(((distance[u]+adjacencyMatrix[u][w])<=distance[w]) && !visited[w])

{

distance[w]=distance[u]+adjacencyMatrix[u][w];

parent[w] = u;

}

}

}

for(int i=2;i<=vertexCount;i++) //Construct Shortest Path Tree

{

shortestPathTree[i][parent[i]] = adjacencyMatrix[i][parent[i]]; shortestPathTree[parent[i]][i] = adjacencyMatrix[parent[i]][i];

}

}

//Returns the closest vertex

int Extract\_Min()

{

int w=1,j;

double min=inf;

for(j=1;j<=vertexCount;j++)

{

if(distance[j]<min && !visited[j])

{

min=distance[j];

w=j;

}

}

return w;

}

//Prints the shortest distances from the root to all other vertices

void printShortestPaths()

{

for(int i=1;i<=vertexCount;i++)

if(i!=root)

cout<<"\n("<<root<<")--("<<i<<"): "<<distance[i];

}

//Prints the Shortest Path Tree

void printShortestPathTree()

{

cout<<"\n";

for (int i = 1; i < vertexCount+1; i++)

{

for (int j = 1; j < vertexCount+1; j++)

{

if(shortestPathTree[i][j] == inf)

cout<<"inf\t";

else

cout<<shortestPathTree[i][j]<<" \t";

}

cout<<"\n";

}

}

//Destructor

~Graph()

{

for (int i = 0; i < vertexCount; i++)

{

delete[] adjacencyMatrix[i];

delete[] MSTMatrix[i];

delete[] shortestPathTree[i];

}

delete[] adjacencyMatrix;

delete[] MSTMatrix;

delete[] shortestPathTree;

delete[] color;

delete[] parent;

}

};

int main()

{

int numOfVertices,numOfEdges;

int u,v;

double w;

string line, sv1,sv2,sw;

int choice;

Graph\* graph;

cout<<"How do you want to input the data?\n 1. Read data from text file.(the file should be located in C:\\ e.g. \"C:\\graph1-Proj2.txt\"\n 2. Manually enter the data\n";

cout<<"Enter choice (1/2): ";

cin>>choice;

switch(choice)

{

case 1:

{

string filename;

cout<<"Enter the name of the file: ";

cin>>filename;

ifstream input;

input.open("C:\\"+filename); //open file

getline(input,line); //read first line

istringstream iss( line );

iss>>numOfVertices;

getline(input,line); //read second line

istringstream iss2( line );

iss2>>numOfEdges;

graph = new Graph(numOfVertices); //create graph

while( getline( input, line ) ) // read each line (u,v,w)

{

istringstream ls( line );

getline(ls,sv1,',');

u = atoi(sv1.c\_str());

getline(ls,sv2,',');

v = atoi(sv2.c\_str());

getline(ls,sw,',');

w = atof(sw.c\_str());

graph->addEdge(u, v, w); //add edge to the graph

}

}

break;

case 2:

{

cout<<"\nEnter no. of vertices : ";

cin>>numOfVertices;

cout<<"\nEnter the no. of edges : ";

cin>>numOfEdges;

graph = new Graph(numOfVertices); //create graph

cout<<"\nEnter the edges : u v w ";

for(int i=0; i<numOfEdges; i++)

{

cout<<"\nEnter edge : ";

cin>>u>>v>>w;

graph->addEdge(u, v, w);

}

}

break;

}

cout<<"\nThe Adjacency Matrix Representation of the Graph is as follows:\n\n";

graph->printAdjacencyMatrix();

cout<<"\nGenerating the Minimum Spanning Tree using Kruskals Algorithm...\n\n";

graph->kruskal();

cout<<"\nThe order of the selected edges are as follows\n";

graph->printMSTOrder();

cout<<"\nThe Minimum Spanning Tree in adjacency matrix representation is as follows\n\n";

graph->printMSTMatrix();

if(graph->isConnected())

{

cout<<"\nGraph is a Connected Graph\n";

cout<<"\nCalculating the shortest path tree from vertex 1(source) using Dijkstra's Algorithm (Priority First Search)...\n";

graph->Dijkstras();

cout<<"\n The shortest distances from the source(vertex 1) to all other vertices are as follows\n";

graph->printShortestPaths();

cout<<"\n\nThe Shortest Path Tree in adjacency matrix representation is as follows\n\n";

graph->printShortestPathTree();

}

else

{

cout<<"\nGraph is Not Connected.\n";

cout<<"Hence it is not possible to find the shortest path tree\n";

}

cin>>choice;

}