ASSIGNMENT 4

1. **Aim** –For a weighted graph G,find the minimum spanning tree using Prims algorithm.

2. **Objective** –To find the minimum spanning tree without any cycles and with the minimum possible total edge weight.

3. **Theory** –

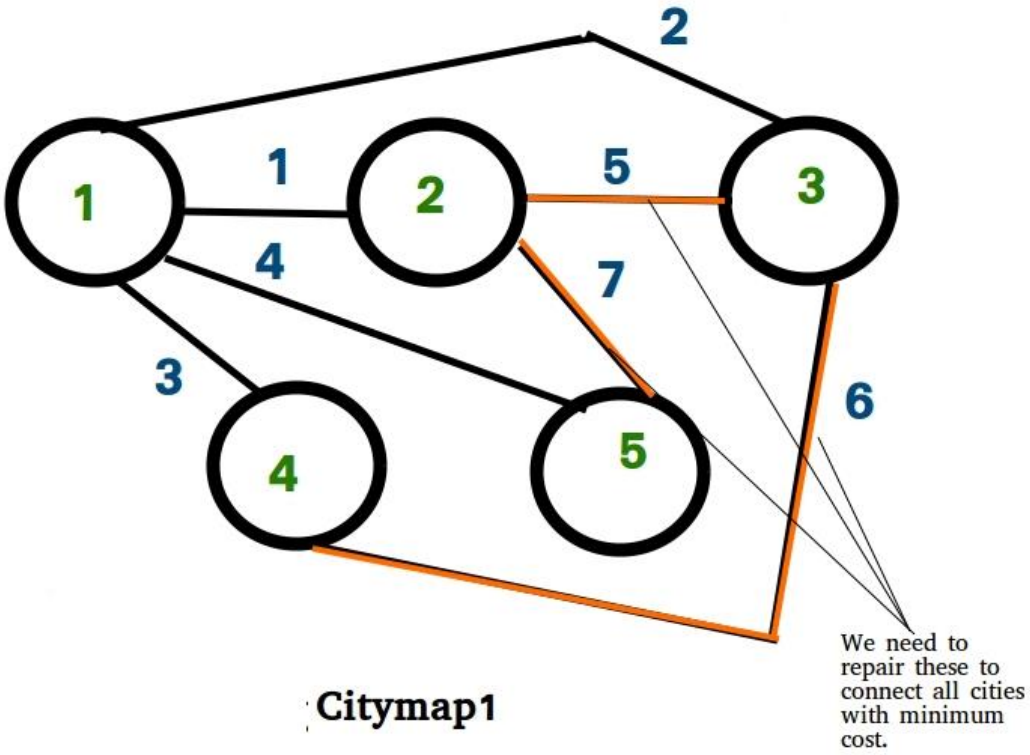
A **minimum spanning tree** (**MST**) or **minimum weight spanning tree** is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible. More generally, any edge-weighted undirected graph (not necessarily connected) has a **minimum spanning forest**, which is a union of the minimum spanning trees for its connected components.

Like Kruskal's algorithm, Prim's algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.
A group of edges that connects two set of vertices in a graph is called cut in graph theory. *So, at every step of Prim's algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the verices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).*

How does Prim's Algorithm Work? The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning* Tree. And they must be connected with the minimum weight edge to make it a *Minimum* Spanning Tree.

There are n cities and there are roads in between some of the cities. Somehow all the roads are damaged simultaneously. We have to repair the roads to connect the cities again. There is a fixed cost to repair a

particular road. Find out the minimum cost to connect all the cities by repairing roads. Input is in matrix(city) form, if city[i][j] = 0 then there is not any road between city i and city j, if city[i][j] = a > 0 then the cost to rebuild the path between city i and city j is a. Print out the minimum cost to connect all the cities.
It is sure that all the cities were connected before the roads were damaged.



Citymap1

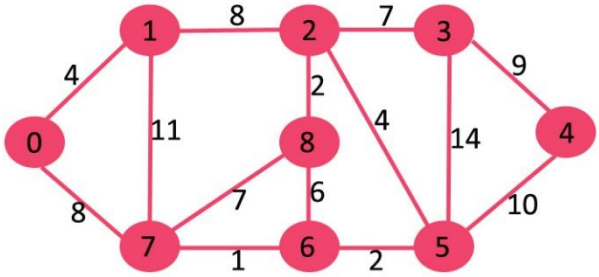
4. **Algorithm** –

1) Create a set *mstSet* that keeps track of vertices already included in MST.
2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
3) While mstSet doesn't include all vertices
….**a)** Pick a vertex *u* which is not there in *mstSet* and has minimum key value.
….**b)** Include *u* to mstSet.
….**c)** Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*,

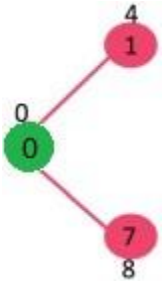
if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*

The idea of using key values is to pick the minimum weight edge from cut. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

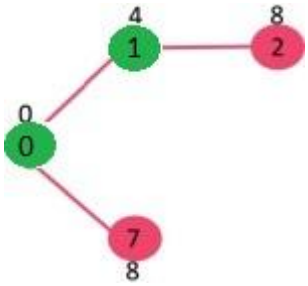
Let us understand with the following example:



The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.

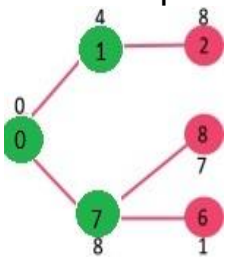


Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.

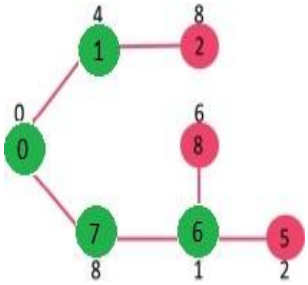


Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (1

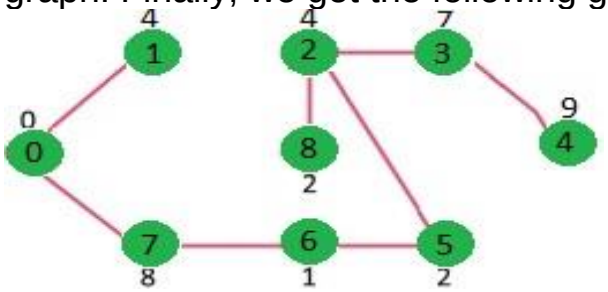
and 7 respectively).



Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.



We repeat the above steps until *mstSet* includes all vertices of given graph. Finally, we get the following graph.



5. **Program code –**

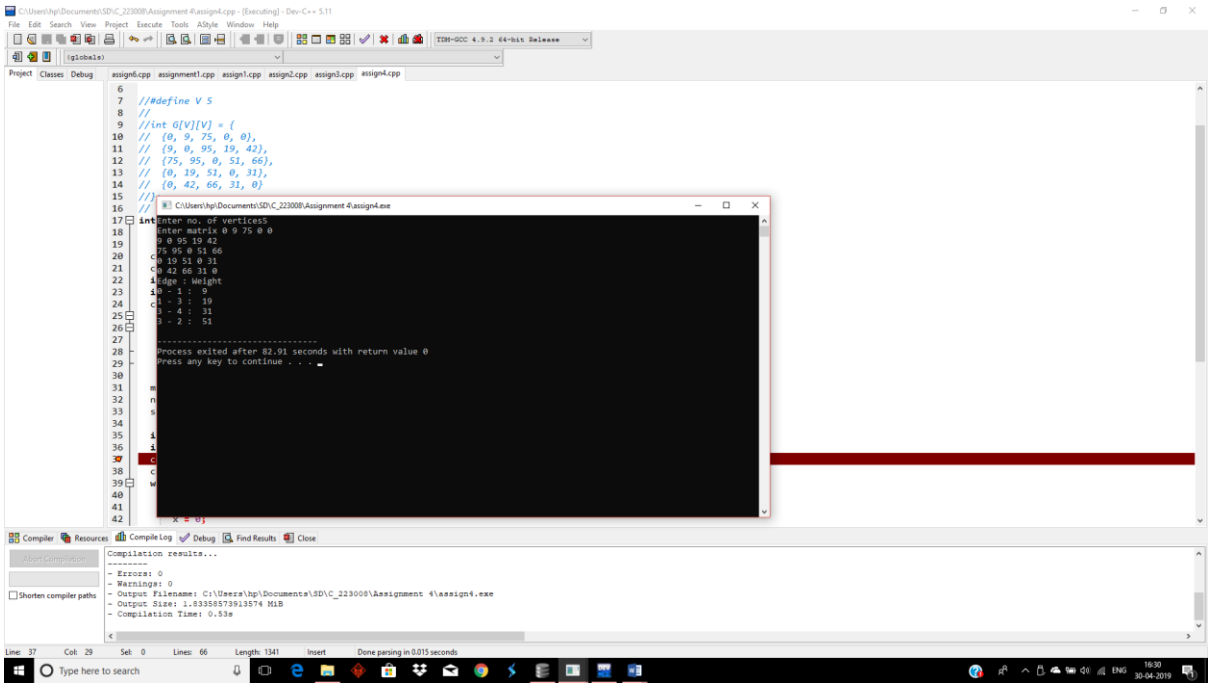
```
#include <iostream>

#include <cstring>

using namespace std;


#define INF 9999999


//#define V 5

//
```

```
//int G[V][V] = {
//  {0, 9, 75, 0, 0},
//  {9, 0, 95, 19, 42},
//  {75, 95, 0, 51, 66},
//  {0, 19, 51, 0, 31},
//  {0, 42, 66, 31, 0}
//};
//
int main () {

   int no_edge,V;
  cout<<"Enter no. of vertices";
  cin>>V;
  int G[V][V];
  int selected[V];
  cout<<"Enter matrix";
   for (int i = 0; i < V; i++) {
          for (int j = 0; j < V; j++) {
                  cin>>G[i][j];
           }
                          }

  memset (selected, false, sizeof (selected));
  no_edge = 0;
  selected[0] = true;
```

```
int x;
int y;
cout << "Edge" << " : " << "Weight";
cout << endl;
while (no_edge < V - 1) {

   int min = INF;
   x = 0;
   y = 0;

   for (int i = 0; i < V; i++) {
    if (selected[i]) {
       for (int j = 0; j < V; j++) {
        if (!selected[j] && G[i][j]) {
           if (min > G[i][j]) {
              min = G[i][j];
              x = i;
              y = j;
           }

        }
     }
    }
   }
   cout << x <<  " - " << y << " :  " << G[x][y];
   cout << endl;
```

```
    selected[y] = true;
    no_edge++;
  }

 return 0;
}
```

6. **Output screen shots/Copy Past From Terminal –**



7. **Conclusion –**

Worst case time complexity of Prim's Algorithm

= O(ElogV) using binary heap

= O(E + VlogV) using Fibonacci heap

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.