PRACTICAL 6

Graph.

PRACTICAL 6.1

AIM:

From a given vertex in a weighted graph, implement a program to find shortest paths to other vertices using Dijkstra's algorithm.

Test Case	Adjacency Matrix of graph	Start Vertex
1	0 1 2 3 4 5 6 7 0 . . . 2 . . . . 1 . . . . . . 7 . 2 . . . . 3 . . . 3 2 . . . . 4 . . 3 . . . 1 7 5 . . . . . . 9 . 6 . 7 . . 1 9 . . 7 . . . . 7 . . .	1
2	0 1 2 3 4 5 6 7 0 . . 2 . . . . . 1 6 . . . . 2 3 8 . . 5 . . . 3 . 9 . . . . . . 4 . . . . . . 1 . 5 . . . 7 . . . . 6 . . 9 . 4 . . 3 7 . . . . . 1 6 .	3

SOFTWARE REQUIREMENT:

C++ Compiler, MS Word.

THEORY:

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years late

The algorithm exists in many variants. Dijkstra's original algorithm found the shortest path between two given nodes but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. For example, if the nodes of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road (for simplicity, ignore red lights, stop signs, toll roads and other obstructions), Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. A widely used application of shortest path algorithms is network routing protocols, most notably IS-IS (Intermediate System to Intermediate System) and Open Shortest Path First (OSPF). It is also employed as a subroutine in other algorithms such as Johnson's.

ALGORITHM:

```
function Dijkstra(Graph, source):

       create vertex set Q

       for each vertex v in Graph:
           dist[v] ← INFINITY
           prev[v] ← UNDEFINED
           add v to Q
       dist[source] ← 0

      while Q is not empty:
          u ← vertex in Q with min dist[u]

          remove u from Q

         for each neighbor v of u still in Q:
              alt ← dist[u] + length(u, v)
              if alt < dist[v]:
                  dist[v] ← alt
                  prev[v] ← u
     return dist[], prev[]
```

CODE:

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int n, m, source;
    cout << "Enter rows and columns: ";
    cin >> n >> m;
    cout << "\nEnter the matrix that contains weights of edges: " << '\n';
    vector<pair<int, int>> g[n + 1]; // 1-indexed adjacency list for of graph
    int mat[n][m];
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            cin >> mat[i][j];
            if (mat[i][j])
            {
                g[i].push_back(make_pair(j, mat[i][j]));
                g[j].push_back(make_pair(i, mat[i][j]));
            }
        }
    }
    cout << "\nEnter source vertex: ";
    cin >> source;

    // Dijkstra's algorithm begins from here
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;
    vector<int> distTo(n + 1, INT_MAX);

    distTo[source] = 0;
    pq.push(make_pair(0, source)); // (dist,from)

    while (!pq.empty())
    {

        pair<int, int> temp = pq.top();
        int dist = temp.first;
        int prev = temp.second;
        pq.pop();

        for (auto it : g[prev])
```

```
        {
            int next = it.first;
            int nextDist = it.second;
            if (distTo[next] > distTo[prev] + nextDist)
            {
                distTo[next] = distTo[prev] + nextDist;
                pq.push(make_pair(distTo[next], next));
            }
        }
    }

    cout << "\nThe shortest distances from source " << source << " are: ";
    for (int i = 1; i <= n; i++)
    {
        if (distTo[i] == INT_MAX)
            cout << "NA" << ' ';
        else
            cout << distTo[i] << ' ';
    }
    cout << "\n";
    return 0;
}
```

OUTPUT:

```
Enter rows and columns: 8 8

Enter the matrix that contains weights of edges:
0 0 0 2 0 0 0 0
0 0 0 0 0 0 7 0
0 0 0 0 3 0 0 0
2 0 0 0 0 0 0 0
0 0 3 0 0 0 1 7
0 0 0 0 0 0 9 0
0 7 0 0 1 9 0 0
0 0 0 0 7 0 0 0

Enter source vertex: 1

The shortest distances from source 1 are: 0 11 NA 8 16 7 15 NA
```

```
Enter rows and columns: 8 8

Enter the matrix that contains weights of edges:
0 0 2 0 0 0 0 0
6 0 0 0 0 0 0 0
3 8 0 0 5 0 0 0
0 9 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 7 0 0 0 0
0 0 9 0 4 0 0 3
0 0 0 0 0 1 6 0

Enter source vertex: 3

The shortest distances from source 3 are: 9 17 0 12 7 11 8 NA
```

CONCLUSION:

In this practical we implemented Dijkstra Algorithm to find shortest paths to other vertices and found out that the time complexity to implement Dijkstra algorithm is $O(ElogV^{)}$ where E is number of edges and V is the number of Vertices.

PRACTICAL 6.2

AIM:

Find Minimum Cost spanning tree of a given undirected graph using Kruskal and Prim's algorithm. Also observe effect on experiment result of choosing those algorithms.

SOFTWARE REQUIREMENT:

C Compiler, MS Word.

THEORY:

Kruskal's algorithm finds a minimum spanning forest of an undirected edge-weighted graph. If the graph is connected, it finds a minimum spanning tree. (A minimum spanning tree of a connected graph is a subset of the edges that forms a tree that includes every vertex, where the sum of the weights of all the edges in the tree is minimized. For a disconnected graph, a minimum spanning forest is composed of a minimum spanning tree for each connected component.) It is a greedy algorithm in graph theory as in each step it adds the next lowest-weight edge that will not form a cycle to the minimum spanning forest.

Prim's algorithm (also known as Jarník's algorithm) is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

Difference between Prim's Algorithm and Kruskal's Algorithm

Prim's Algorithm	**Kruskal's Algorithm**
It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm has a time complexity of O(V2), V being the number of vertices and can be improved up to O(E log V) using Fibonacci heaps.	Kruskal's algorithm's time complexity is O(E log V), V being the number of vertices.
Prim's algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest (disconnected components) at any instant as well as it can work on disconnected components
Prim's algorithm runs faster in dense graphs.	Kruskal's algorithm runs faster in sparse graphs.

Prim's algorithm uses List Data Structure.	Kruskal's algorithm uses Heap Data Structure.

ALGORITHM:

Kruskal Algorithm:

```
algorithm Kruskal(G) is
    F:= ∅
    for each v ∈ G.V do
        MAKE-SET(v)
    for each (u, v) in G.E ordered by weight(u, v), increasing do
        if FIND-SET(u) ≠ FIND-SET(v) then
            F:= F ∪ {(u, v)} ∪ {(v, u)}
            UNION(FIND-SET(u), FIND-SET(v))
    return F
```

Prim's Algorithm:

prim's algorithm shows how we create two sets of vertices U and V-U. U contains the list of vertices that have been visited and V-U the list of vertices that haven't. One by one, we move vertices from set V-U to set U by connecting the least weight edge.

```
T = ∅;
U = { 1 };
while (U ≠ V)
    let (u, v) be the lowest cost edge such that u ∈ U and v ∈ V - U;
    T = T ∪ {(u, v)}
    U = U ∪ {v}
```

CODE:

Kruskal Algorithm:

```
#include <bits/stdc++.h>
using namespace std;

// Union Find Algo
vector<int> p(100005), r(100005);
void make_set(int v)
{
    for (int i = 0; i < v; ++i)
        p[i] = i;
    p[v] = v;
    r[v] = 1;
}

int find(int node)
{
    if (p[node] == node)
        return node;
    return p[node] = find(p[node]);
}

void union_sets(int u, int v)
{

    int a = find(u);
    int b = find(v);
    if (r[u] < r[v])
    {
        p[a] = b;
        r[v] += r[u];
    }
    else
    {
        p[a] = b;
        r[u] += r[v];
    }
}

int main()
{
    int n, m;
    cout << "Enter rows and columns: ";
    cin >> n >> m; // vertices & edges
```

```
    int x, y, wt;
    make_set(n);
    cout << "\nEnter the matrix that contains weights of edges: " << '\n';
    vector<pair<int, pair<int, int>>> krskl;
    for (int i = 0; i < m; i++)
    {
        cin >> x >> y >> wt;
        krskl.push_back({wt, {x, y}}); // making pair of pair of vector
[pair2:vertexes] [pair1:weight&pair2]
    }
    cout << krskl.size() << '\n';
    sort(krskl.begin(), krskl.end()); //  sorting according to weights of edges
    int cost = 0;

    vector<pair<int, int>> mst;

    for (int i = 0; i < m; i++)
    {
        int x = krskl[i].second.first;
        int y = krskl[i].second.second;
        int c = krskl[i].first;

        if (find(x) != find(y))
        { // if in not in same set then, include in MST & make it in one set
            cost += c;
            mst.push_back({x, y});
            union_sets(x, y);
        }
    }
    cout << "\nMST : "
         << "\n";
    for (auto x : mst)
    {
        cout << x.first << " - " << x.second << '\n';
    }
    cout << "\nMinimum cost: " << cost << '\n';
}
```

Prim's Algorithm

```
#include <bits/stdc++.h>
using namespace std;

int main()
{

    int n, m;
    cout << "Enter rows and columns: ";
    cin >> n >> m;
    vector<pair<int, int>> adj[n]; // array of vector(pair)

    cout << "\nEnter the matrix that contains weights of edges: " << '\n';
    for (int i = 0; i < m; i++)
    {
        int x, y, wt;
        cin >> x >> y >> wt;
        adj[x].push_back({y, wt}); // pair of adjacent element and weight
        adj[y].push_back({x, wt}); // pair of adjacent element and weight
    }

    int parent[n];   // for printing the spanning tree
    int dist[n];     // for count the minimum cost
    bool visited[n]; // to determine if this is already visited or not, to avoid
cycle

    for (int i = 0; i < n; i++)
    {
        dist[i] = INT_MAX;
        visited[i] = false;
    }

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;

    pq.push({0, 0}); // distance , node
    dist[0] = 0;
    parent[0] = -1;

    while (!pq.empty())
    {

        int node = pq.top().second;
        visited[node] = true; // node visited to avoid cycle
```

```
        pq.pop();

        for (auto x : adj[node])
        {

            int adj_vertex = x.first;
            int wtt = x.second;

            // if not visited & weight is lesser than other adjcent vertex,
update else continue
            if (visited[adj_vertex] == false && wtt < dist[adj_vertex])
            {
                parent[adj_vertex] = node;
                dist[adj_vertex] = wtt;
                pq.push({dist[adj_vertex], adj_vertex}); //  pushing the new pair
with min distance
            }
            else
                continue;
        }
    }

    int cost = 0;
    cout << endl << "MST : "
         << "\n";
    for (int i = 1; i < n; i++)
    {
        cost += dist[i];
        cout << parent[i] << " - " << i << " \n";
    }
    cout << "\nMinimum cost: " << cost << '\n';
}
```

OUTPUT:

Kruskal Algorithm

```
Enter rows and columns: 6 9

Enter the matrix that contains weights of edges:
5 4 9
5 1 4
4 1 1
4 3 5
4 2 3
1 2 2
3 2 3
3 6 8
2 6 8
9

MST :
4 - 1
1 - 2
3 - 2
5 - 1
2 - 6

Minimum cost: 18
```

Prim's Algorithm

```
Enter rows and columns: 5 6

Enter the matrix that contains weights of edges:
0 3 6
0 1 2
3 1 8
1 2 3
1 4 5
2 4 1

MST :
0 - 1
1 - 2
0 - 3
2 - 4

Minimum cost: 12
```

CONCLUSION

In this practical we implemented Kruskal's and Prim's Algorithm to Find Minimum Cost spanning tree of a given undirected graph and found out the time complexity of both the algorithms.

i. Kruskal's Algorithm: O(ElogV)
ii. Prim's Algorithm: O((V+E)logV)

Where E is number of edges and V is the number of Vertices.