Module 7 - Graphs

1) Create a graph storage structure using Adjacency Matrix

```
#include<iostream>

using namespace std;

int vertArr[20][20]; //the adjacency matrix initially 0

int count = 0;

void displayMatrix(int v) {

int i, j;

for(i = 0; i < v; i++) {

for(j = 0; j < v; j++) {

cout << vertArr[i][j] << " ";

}

cout << endl;

}

}

void add_edge(int u, int v) { //function to add edge into the matrix

vertArr[u][v] = 1;

vertArr[v][u] = 1;

}

main(int argc, char* argv[]) {

int v = 6; //there are 6 vertices in the graph

add_edge(0, 4);

add_edge(0, 3);

add_edge(1, 2);

add_edge(1, 4);

add_edge(1, 5);

add_edge(2, 3);

add_edge(2, 5);

add_edge(5, 3);

add_edge(5, 4);
displayMatrix(v);
}
```

Output:

```
0 0 0 1 1 0
0 0 1 0 1 1
0 1 0 1 0 1
1 0 1 0 0 1
11 0 0 0 1
0 1 1 1 1 0
```

2) Create a minimum spanning tree using – Kruskal's algorithm

```
#include<bits/stdc++.h>
using namespace std;
typedef pair<int, int> iPair;
struct Graph
{
int V, E;
vector< pair<int, iPair> > edges;
Graph(int V, int E)
{
this->V = V;
this->E = E;
}
void addEdge(int u, int v, int w)
{
edges.push_back({w, {u, v}});
}
int kruskalMST();
};
struct DisjointSets
{
```

```
int *parent, *rnk;
int n;
DisjointSets(int n)
{
this->n = n;
parent = new int[n+1];
rnk = new int[n+1];
for (int i = 0; i <= n; i++)
{
rnk[i] = 0;
parent[i] = i;
}
}
int find(int u)
{
if (u != parent[u])
parent[u] = find(parent[u]);
return parent[u];
}
void merge(int x, int y)
{
x = find(x), y = find(y);
if (rnk[x] > rnk[y])
parent[y] = x;
else
parent[x] = y;
if (rnk[x] == rnk[y])
rnk[y]++;
}
};
int Graph::kruskalMST()
{
```

```
int mst_wt = 0;
sort(edges.begin(), edges.end());
DisjointSets ds(V);
vector< pair<int, iPair> >::iterator it;
for (it=edges.begin(); it!=edges.end(); it++)
{
int u = it->second.first;
int v = it->second.second;
int set_u = ds.find(u);
int set_v = ds.find(v);
if (set_u != set_v)
{
cout << u << " - " << v << endl;
mst_wt += it->first;
ds.merge(set_u, set_v);
}
}
return mst_wt;
}
int main()
{
int V = 9, E = 14;
Graph g(V, E);
g.addEdge(0, 1, 4);
g.addEdge(0, 7, 8);
g.addEdge(1, 2, 8);
g.addEdge(1, 7, 11);
g.addEdge(2, 3, 7);
g.addEdge(2, 8, 2);
g.addEdge(2, 5, 4);
g.addEdge(3, 4, 9);
g.addEdge(3, 5, 14);
```

```
g.addEdge(4, 5, 10);
g.addEdge(5, 6, 2);
g.addEdge(6, 7, 1);
g.addEdge(6, 8, 6);
g.addEdge(7, 8, 7);
cout << "Edges of MST are \n";
int mst_wt = g.kruskalMST();
cout << "\nWeight of MST is " << mst_wt;
return 0;
}
```

Output:

```
Edges of MST are
6 - 7
2 - 8
5 - 6
0 - 1
2 - 5
2 - 3
0 - 7
3 - 4
Weight of MST is 37
--------------------------------
```

3) Implementation of graph traversal (DFS & BFS)

Breadth First Search

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int cost[10][10],i,j,k,n,qu[10],front,rare,v,visit[10],visited[10];
int main()
{
int m;
cout <<"Enter no of vertices:";
cin >> n;
cout <<"Enter no of edges:";
cin >> m;
cout <<"\nEDGES \n";
for(k=1; k<=m; k++)
{
cin >>i>>j;
cost[i][j]=1;
}
cout <<"Enter initial vertex to traverse from:";
cin >>v;
cout <<"Visitied vertices:";
cout <<v<<" ";
visited[v]=1;
k=1;
while(k<n)
{
for(j=1; j<=n; j++)
if(cost[v][j]!=0 && visited[j]!=1 && visit[j]!=1)
{
visit[j]=1;
qu[rare++]=j;
}
v=qu[front++];
cout<<v <<" ";
k++;
visit[v]=0;
visited[v]=1;
}
return 0;
}
```

Output

```
Enter no of vertices:7
Enter no of edges:9
EDGES
0 1
0 3
1 5
1 3
1 6
3 2
3 4
4 6
2 5
Enter initial vertex to traverse from:0
Visitied vertices:0 1 3 5 6 2 4
```

Depth First Search

Program

```
#include<iostream>
#include<conio.h>
#include<stdlib.h>
using namespace std;
int cost[10][10],i,j,k,n,stk[10],top,v,visit[10],visited[10];
int main()
{
int m;
cout <<"Enter no of vertices:";
cin >> n;
cout <<"Enter no of edges:";
cin >> m;
cout <<"\nEDGES \n";
for(k=1; k<=m; k++)
{
cin >>i>>j;
cost[i][j]=1;
}
cout <<"Enter initial vertex to traverse from:";
cin >>v;
cout <<"DFS ORDER OF VISITED VERTICES:";
cout << v <<" ";
visited[v]=1;
k=1;
while(k<n)
{
for(j=n; j>=1; j--)
if(cost[v][j]!=0 && visited[j]!=1 && visit[j]!=1)
{
visit[j]=1;
stk[top]=j;
top++;
}
v=stk[--top];
cout<<v << " ";
k++;
visit[v]=0;
visited[v]=1;
}
return 0;
}
```

Output

```
Enter no of vertices:7
Enter no of edges:9
EDGES
0 1
```

```
0 3
1 5
1 3
1 6
3 2
3 4
4 6
2 5
Enter initial vertex to traverse from:0
DFS ORDER OF VISITED VERTICES:0 1 5 6 3 2 4
```