# TASK 2: KRUSKAL’S ALGORITHM:

#include<iostream>

#include<string.h>

using namespace std;

class Graph

{

char vertices[10][10];

int cost[10][10], no, edges;

public:

Graph();

void creat\_graph();

void display();

int Position(char[]);

void kruskal\_algo();

};

/\* Initialzing adj matrix with 999 \*/

/\* 999 denotes infinite distance \*/

Graph::Graph()

{

no=0;

for(int i=0;i<10;i++)

for(int j=0;j<10;j++)

{

cost[i][j]=999;

}

}

/\* Taking inputs for creating graph \*/

void Graph::creat\_graph()

{

char ans,Start[10],End[10];

int wt,i,j;

cout<<"Enter the number of vertices: ";

cin>>no;

cout << "Enter the number of edges: ";

cin >> edges;

cout<<"\nEnter the vertices: ";

for(i=0;i<no;i++)

cin>>vertices[i];

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

{

cout << "\nEnter Start and End vertex of the edge: ";

cin >> Start >> End;

cout << "Enter weight: ";

cin >> wt;

i = Position(Start);

j = Position(End);

cost[i][j] = cost[j][i] = wt;

}

}

/\* Displaying Cost matrix \*/

void Graph::display()

{

int i,j;

cout<<"\n\nCost matrix: ";

for(i=0;i<no;i++)

{

cout<<"\n";

for(j=0;j<no;j++)

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

}

}

/\* Retrieving position of vertices in 'vertices' array \*/

int Graph::Position(char key[10])

{

int i;

for(i=0;i<10;i++)

if(strcmp(vertices[i],key)==0)

return i;

return -1;

}

void Graph::kruskal\_algo()

{

int i,j,v[10]={0},x,y,Total\_cost=0,min,gr=1,flag=0,temp,d;

while(flag==0)

{

min=999;

for(i=0;i<no;i++)

{

for(j=0;j<no;j++)

{

if(cost[i][j]<min)

{

min=cost[i][j];

x=i;

y=j;

}

}

}

if(v[x]==0 && v[y]==0)

{

v[x]=v[y]=gr;

gr++;

}

else if(v[x]!=0 && v[y]==0)

v[y]=v[x];

else if(v[x]==0 && v[y]!=0)

v[x]=v[y];

else

{

if(v[x]!=v[y])

{

d=v[x];

for(i=0;i<no;i++)

{

if(v[i]==d)

v[i]=v[y];

}//end for

}

}

cost[x][y]=cost[y][x]=999;

Total\_cost=Total\_cost+min; /\* calculating cost of minimum spanning tree \*/

cout<<"\n\t"<<vertices[x]<<"\t\t"<<vertices[y]<<"\t\t"<<min;

temp=v[0]; flag=1;

for(i=0;i<no;i++)

{

if(temp!=v[i])

{

flag=0;

break;

}

}

}

cout<<"\nTotal cost of the tree= "<<Total\_cost;

}

int main()

{

Graph g;

g.creat\_graph();

g.display();

cout<<"\n\n\nMinimum Spanning tree using kruskal algo=>";

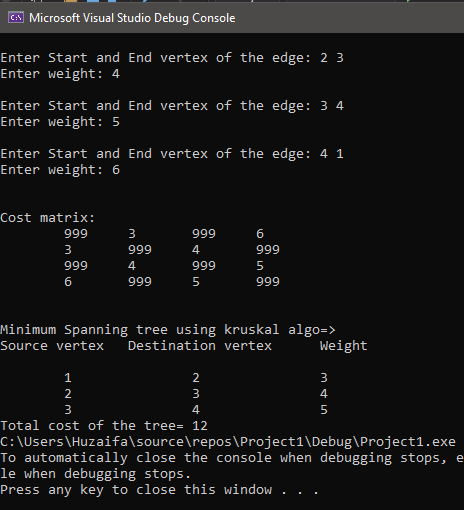
cout<<"\nSource vertex\tDestination vertex\tWeight\n";

g.kruskal\_algo();

return 0;

}

# OUTPUT:



# TASK 3:

# PRIM’S ALGORITHM:

#include<iostream>

using namespace std;

class graph

{

public:

int graph[20][20], n, e;

void node();

void display();

void prims();

};

void graph::node()

{

int src, dest, cost;

cout << "Enter total vertices: "; //Taking input

cin >> n;

cout << "Enter total edges: ";

cin >> e;

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

{

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

{

graph[i][j] = 0; //Initialzing with zero

}

}

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

{

cout << "Enter source, Destination and Cost: "; //Setting cost to their edges

cin >> src >> dest >> cost;

graph[src][dest] = cost;

graph[dest][src] = cost;

}

}

void graph::display() // Output Function

{

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

{

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

{

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

}

cout << endl;

}

}

void graph::prims()

{

int visited[20], min, src, dest, total = 0, m = 0;

int key[20], parent[20];

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

{

key[i] = 99999; // Initialize key values as infinity

visited[i] = 0; // Mark all vertices as not visited

}

cout << "Source for finding Minimum Spanning Tree: "; //Stating point for algo

cin >> src;

key[src] = 0; // Make key value of the source vertex as 0 so that it is picked first

parent[src] = -1; // First node is always root of MST

for (int i = 1; i <= n - 1; i++)

{

// Pick the minimum key vertex from the set of vertices not yet included in MST

min = 99999;

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

{

if (visited[j] == 0 && key[j] < min)

{

min = key[j];

m = j;

}

}

visited[m] = 1; // Add the picked vertex to the MST set

// Update key value and parent index of the adjacent vertices of the picked vertex

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

{

if (graph[m][j] && visited[j] == 0 && graph[m][j] < key[j])

{

parent[j] = m;

key[j] = graph[m][j];

}

}

}

// Print the edges of the minimum spanning tree

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

{

cout << "Edge= " << parent[i] << "..." << i << " cost=" << graph[i][parent[i]] << endl;

total += graph[i][parent[i]];

}

cout << "Total cost of minimum spanning tree: " << total;

}

int main()

{

graph g;

g.node();

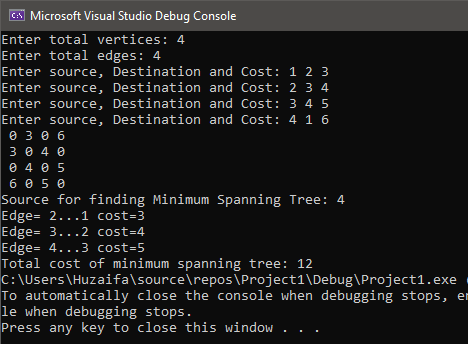
g.display();

g.prims();

return 0;

}

# OUTPUT:



# TASK 4:

# BFS:

#include <iostream>

#include<queue>

using namespace std;

class Graph

{

private:

int Matrix[30][30];

int Vertices;

public:

Graph(int num)

{

if (num <= 30)

{

Vertices = num;

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

{

for (int j = 0; j <= Vertices; j++)

{

Matrix[i][j] = 0;

}

}

}

else

{

cout << "Size is exceeding" << endl;

}

}

void addEdge(int i, int j)

{

Matrix[i][j] = 1;

Matrix[j][i] = 1;

}

void removeEdge(int i, int j)

{

Matrix[i][j] = 0;

Matrix[j][i] = 0;

}

void print()

{

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

{

cout << i << ":" << " ";

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

{

if (Matrix[i][j] == 1)

{

cout << j << ",";

}

}

cout << endl;

}

}

void BFS(int start)

{

bool\* visited = new bool[Vertices + 1];

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

{

visited[i] = false;

}

queue<int> q;

visited[start] = true;

q.emplace(start);

while (!q.empty())

{

int v = q.front();

q.pop();

cout << v << " ";

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

{

if (Matrix[v][i] && !visited[i])

{

visited[i] = true;

q.emplace(i);

}

}

}

delete[] visited;

}

};

int main()

{

Graph g(5);

g.addEdge(1, 2);

g.addEdge(1, 3);

g.addEdge(1, 4);

g.addEdge(1, 5);

g.addEdge(2, 1);

g.addEdge(2, 4);

g.addEdge(3, 1);

g.addEdge(3, 5);

g.addEdge(3, 4);

g.addEdge(4, 1);

g.addEdge(4, 2);

g.addEdge(4, 3);

g.addEdge(5, 1);

g.addEdge(5, 3);

g.print();

int vertex;

cout << "\nEnter vertex to start to start DFS from: " << endl;

cin >> vertex;

cout << "DFS starting from: " << vertex << endl;

g.BFS(vertex);

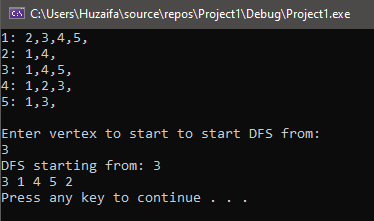
cout << endl;

system("pause");

return 0;

}

# OUTPUT:



# TASK 5:

# DFS:

#include <iostream>

#include<stack>

using namespace std;

class Graph

{

private:

int Matrix[30][30];

int Vertices;

public:

Graph(int num)

{

if (num <= 30)

{

Vertices = num;

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

{

for (int j = 0; j <= Vertices; j++)

{

Matrix[i][j] = 0;

}

}

}

else

{

cout << "Size is exceeding" << endl;

}

}

void addEdge(int i, int j)

{

Matrix[i][j] = 1;

Matrix[j][i] = 1;

}

void removeEdge(int i, int j)

{

Matrix[i][j] = 0;

Matrix[j][i] = 0;

}

void print()

{

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

{

cout << i << ":" << " ";

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

{

if (Matrix[i][j] == 1)

{

cout << j << ",";

}

}

cout << endl;

}

}

void DFS(int start)

{

bool\* visited = new bool[Vertices + 1];

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

{

visited[i] = false;

}

stack<int> s;

s.push(start);

while (!s.empty())

{

int v = s.top();

s.pop();

if (!visited[v])

{

visited[v] = true;

cout << v << " ";

for (int i = Vertices; i >= 1; i--)

{

if (Matrix[v][i] && !visited[i])

{

s.push(i);

}

}

}

}

delete[] visited;

}

};

int main()

{

Graph g(5);

g.addEdge(1, 2);

g.addEdge(1, 3);

g.addEdge(1, 4);

g.addEdge(1, 5);

g.addEdge(2, 1);

g.addEdge(2, 4);

g.addEdge(3, 1);

g.addEdge(3, 5);

g.addEdge(3, 4);

g.addEdge(4, 1);

g.addEdge(4, 2);

g.addEdge(4, 3);

g.addEdge(5, 1);

g.addEdge(5, 3);

g.print();

int vertex;

cout << "\nEnter vertex to start to start DFS from: " << endl;

cin >> vertex;

cout << "DFS starting from: " << vertex << endl;

g.DFS(vertex);

cout << endl;

system("pause");

return 0;

}

# OUTPUT:

