Lab #15

Task #1:

#include <iostream>

using namespace std;

const int INF = 1e9; // A large constant value representing infinity

// Function to find the vertex with the minimum distance value,

// from the set of vertices not yet included in the shortest path

int minDistance(int dist[], bool sptSet[], int V)

{

int min = INF, min\_index;

for (int v = 0; v < V; v++) {

if (!sptSet[v] && dist[v] <= min)

{

min = dist[v];

min\_index = v;

}

}

return min\_index;

}

// Function to print the array that stores the shortest distance from the source.

void printSolution(int dist[], int V) {

cout << "Vertex \t Distance from Source" << endl;

for (int i = 0; i < V; i++) {

cout << i << "\t" << dist[i] << endl;

}

}

void dijkstra(int graph[][9], int src, int V) {

int dist[9]; // Array to store the shortest distance from the source.

bool sptSet[9]; // Array to keep track of vertices included in the shortest path tree.

// Initialize distance and sptSet arrays

for (int i = 0; i < V; i++) {

dist[i] = INF;

sptSet[i] = false;

}

dist[src] = 0; // Distance from source to itself is 0

for (int count = 0; count < V - 1; count++) {

int u = minDistance(dist, sptSet, V);

sptSet[u] = true;

for (int v = 0; v < V; v++) {

if (!sptSet[v] && graph[u][v] && dist[u] != INF && (dist[u] + graph[u][v] < dist[v])) {

dist[v] = dist[u] + graph[u][v];

}

}

}

// Print the shortest distance from the source to all vertices

printSolution(dist, V);

}

int main() {

const int V = 9; // Number of vertices in the graph

int graph[V][9] = {

{ 0, 4, 0, 0, 0, 0, 0, 8, 0 },

{ 4, 0, 8, 0, 0, 0, 0, 11, 0 },

{ 0, 8, 0, 7, 0, 4, 0, 0, 2 },

{ 0, 0, 7, 0, 9, 14, 0, 0, 0 },

{ 0, 0, 0, 9, 0, 10, 0, 0, 0 },

{ 0, 0, 4, 14, 10, 0, 2, 0, 0 },

{ 0, 0, 0, 0, 0, 2, 0, 1, 6 },

{ 8, 11, 0, 0, 0, 0, 1, 0, 7 },

{ 0, 0, 2, 0, 0, 0, 6, 7, 0 }

};

int source = 0; // Source vertex

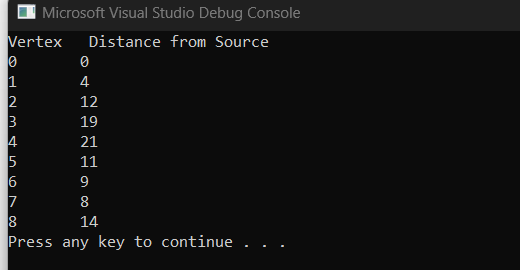
dijkstra(graph, source, V);

system("pause");

return 0;

}

Output:



Task #2:

#include <iostream>

using namespace std;

const int V = 5; // Number of vertices in the graph

// Function to find the vertex with the minimum key value,

// from the set of vertices not yet included in the MST.

int minKey(int key[], bool mstSet[]) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++) {

if (!mstSet[v] && key[v] < min) {

min = key[v];

min\_index = v;

}

}

return min\_index;

}

// Function to print the MST.

void printMST(int parent[], int graph[V][V]) {

cout << "Edge \tWeight" << endl;

for (int i = 1; i < V; i++) {

cout << parent[i] << " - " << i << "\t" << graph[i][parent[i]] << endl;

}

}

void primMST(int graph[V][V]) {

int parent[V]; // Array to store the MST.

int key[V]; // Key values used to pick the minimum weight edge.

bool mstSet[V]; // Array to keep track of vertices included in MST.

// Initialize key values and mstSet arrays

for (int i = 0; i < V; i++) {

key[i] = INT\_MAX;

mstSet[i] = false;

}

key[0] = 0; // Start with the first vertex

for (int count = 0; count < V - 1; count++) {

int u = minKey(key, mstSet);

mstSet[u] = true;

for (int v = 0; v < V; v++) {

if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {

parent[v] = u;

key[v] = graph[u][v];

}

}

}

// Print the MST

printMST(parent, graph);

}

// Data structure to represent an edge

struct Edge {

int src, dest, weight;

};

// Function to find the parent of a vertex in the set

int findParent(int v, int parent[]) {

if (parent[v] == v)

return v;

return findParent(parent[v], parent);

}

// Function to perform union of two sets

void unionSets(int u, int v, int parent[]) {

int parent\_u = findParent(u, parent);

int parent\_v = findParent(v, parent);

parent[parent\_u] = parent\_v;

}

void kruskalMST(int graph[V][V]) {

Edge edges[V \* V]; // Array to store all the edges

int edgeCount = 0;

// Store all the edges of the graph in the array

for (int i = 0; i < V; i++) {

for (int j = i + 1; j < V; j++) {

if (graph[i][j]) {

edges[edgeCount].src = i;

edges[edgeCount].dest = j;

edges[edgeCount].weight = graph[i][j];

edgeCount++;

}

}

}

// Sort the edges based on weight using Bubble Sort (can be replaced with a more efficient sorting algorithm)

for (int i = 0; i < edgeCount - 1; i++) {

for (int j = 0; j < edgeCount - i - 1; j++) {

if (edges[j].weight > edges[j + 1].weight) {

Edge temp = edges[j];

edges[j] = edges[j + 1];

edges[j + 1] = temp;

}

}

}

Edge mst[V - 1]; // Array to store the MST edges

int mstCount = 0;

int parent[V]; // Array to store the parent of each vertex

for (int i = 0; i < V; i++) {

parent[i] = i; // Initialize each vertex as its own parent

}

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

int u = edges[i].src;

int v = edges[i].dest;

int weight = edges[i].weight;

int parent\_u = findParent(u, parent);

int parent\_v = findParent(v, parent);

if (parent\_u != parent\_v) {

mst[mstCount] = edges[i];

mstCount++;

unionSets(parent\_u, parent\_v, parent);

}

}

// Print the MST

cout << "Edge \tWeight" << endl;

for (int i = 0; i < V - 1; i++) {

cout << mst[i].src << " - " << mst[i].dest << "\t" << mst[i].weight << endl;

}

}

int main() {

int graph[V][V] =

{

{ 0, 2, 0, 6, 0 },

{ 2, 0, 3, 8, 5 },

{ 0, 3, 0, 0, 7 },

{ 6, 8, 0, 0, 9 },

{ 0, 5, 7, 9, 0 }

};

cout << "Minimum Spanning Tree using Prim's algorithm:" << endl;

primMST(graph);

cout << "\nMinimum Spanning Tree using Kruskal's algorithm:" << endl;

kruskalMST(graph);

system("pause");

return 0;

}

Output:

