**EXPERIMENT 5**

**TITLE:** GRAPH ALGORITHMS

1. **Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal’s algorithm**

**Code:**

// C++ program for Kruskal's algorithm to find Minimum Spanning Tree

// of a given connected, undirected and weighted graph

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// a structure to represent a weighted edge in graph

struct Edge

{

int src, dest, weight;

};

// a structure to represent a connected, undirected

// and weighted graph

struct Graph

{

// V-> Number of vertices, E-> Number of edges

int V, E;

// graph is represented as an array of edges.

// Since the graph is undirected, the edge

// from src to dest is also edge from dest

// to src. Both are counted as 1 edge here.

struct Edge\* edge;

};

// Creates a graph with V vertices and E edges

struct Graph\* createGraph(int V, int E)

{

struct Graph\* graph = new Graph;

graph->V = V;

graph->E = E;

graph->edge = new Edge[E];

return graph;

}

// A structure to represent a subset for union-find

struct subset

{

int parent;

int rank;

};

// A utility function to find set of an element i

// (uses path compression technique)

int find(struct subset subsets[], int i)

{

// find root and make root as parent of i

// (path compression)

if (subsets[i].parent != i)

subsets[i].parent = find(subsets, subsets[i].parent);

return subsets[i].parent;

}

// A function that does union of two sets of x and y

// (uses union by rank)

void Union(struct subset subsets[], int x, int y)

{

int xroot = find(subsets, x);

int yroot = find(subsets, y);

// Attach smaller rank tree under root of high

// rank tree (Union by Rank)

if (subsets[xroot].rank < subsets[yroot].rank)

subsets[xroot].parent = yroot;

else if (subsets[xroot].rank > subsets[yroot].rank)

subsets[yroot].parent = xroot;

// If ranks are same, then make one as root and

// increment its rank by one

else

{

subsets[yroot].parent = xroot;

subsets[xroot].rank++;

}

}

// Compare two edges according to their weights.

// Used in qsort() for sorting an array of edges

int myComp(const void\* a, const void\* b)

{

struct Edge\* a1 = (struct Edge\*)a;

struct Edge\* b1 = (struct Edge\*)b;

return a1->weight > b1->weight;

}

// The main function to construct MST using Kruskal's algorithm

void KruskalMST(struct Graph\* graph)

{

int V = graph->V;

struct Edge result[V]; // Tnis will store the resultant MST

int e = 0; // An index variable, used for result[]

int i = 0; // An index variable, used for sorted edges

// Step 1: Sort all the edges in non-decreasing

// order of their weight. If we are not allowed to

// change the given graph, we can create a copy of

// array of edges

qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

// Allocate memory for creating V ssubsets

struct subset \*subsets =

(struct subset\*) malloc( V \* sizeof(struct subset) );

// Create V subsets with single elements

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

{

subsets[v].parent = v;

subsets[v].rank = 0;

}

// Number of edges to be taken is equal to V-1

while (e < V - 1)

{

// Step 2: Pick the smallest edge. And increment

// the index for next iteration

struct Edge next\_edge = graph->edge[i++];

int x = find(subsets, next\_edge.src);

int y = find(subsets, next\_edge.dest);

// If including this edge does't cause cycle,

// include it in result and increment the index

// of result for next edge

if (x != y)

{

result[e++] = next\_edge;

Union(subsets, x, y);

}

// Else discard the next\_edge

}

// print the contents of result[] to display the

// built MST

printf("Following are the edges in the constructed MST\n");

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

printf("%d -- %d == %d\n", result[i].src, result[i].dest,

result[i].weight);

return;

}

// Driver program to test above functions

int main()

{

/\* Let us create following weighted graph

10

0--------1

| \ |

6| 5\ |15

| \ |

2--------3

4 \*/

int V = 4; // Number of vertices in graph

int E = 5; // Number of edges in graph

struct Graph\* graph = createGraph(V, E);

// add edge 0-1

graph->edge[0].src = 0;

graph->edge[0].dest = 1;

graph->edge[0].weight = 10;

// add edge 0-2

graph->edge[1].src = 0;

graph->edge[1].dest = 2;

graph->edge[1].weight = 6;

// add edge 0-3

graph->edge[2].src = 0;

graph->edge[2].dest = 3;

graph->edge[2].weight = 5;

// add edge 1-3

graph->edge[3].src = 1;

graph->edge[3].dest = 3;

graph->edge[3].weight = 15;

// add edge 2-3

graph->edge[4].src = 2;

graph->edge[4].dest = 3;

graph->edge[4].weight = 4;

KruskalMST(graph);

return 0;

}

1. **Find Minimum Cost Spanning Tree of a given undirected graph using prim's algorithm**

**Code:**

// A C / C++ program for Prim's Minimum

// Spanning Tree (MST) algorithm. The program is

// for adjacency matrix representation of the graph

#include <stdio.h>

#include <limits.h>

#include<stdbool.h>

// Number of vertices in the graph

#define V 5

// A utility function to find the vertex with

// minimum key value, from the set of vertices

// not yet included in MST

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

{

// Initialize min value

int min = INT\_MAX, min\_index;

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

if (mstSet[v] == false && key[v] < min)

min = key[v], min\_index = v;

return min\_index;

}

// A utility function to print the

// constructed MST stored in parent[]

int printMST(int parent[], int n, int graph[V][V])

{

printf("Edge \tWeight\n");

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

printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);

}

// Function to construct and print MST for

// a graph represented using adjacency

// matrix representation

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

{

// Array to store constructed MST

int parent[V];

// Key values used to pick minimum weight edge in cut

int key[V];

// To represent set of vertices not yet included in MST

bool mstSet[V];

// Initialize all keys as INFINITE

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

key[i] = INT\_MAX, mstSet[i] = false;

// Always include first 1st vertex in MST.

// Make key 0 so that this vertex is picked as first vertex.

key[0] = 0;

parent[0] = -1; // First node is always root of MST

// The MST will have V vertices

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

{

// Pick the minimum key vertex from the

// set of vertices not yet included in MST

int u = minKey(key, mstSet);

// Add the picked vertex to the MST Set

mstSet[u] = true;

// Update key value and parent index of

// the adjacent vertices of the picked vertex.

// Consider only those vertices which are not

// yet included in MST

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

// graph[u][v] is non zero only for adjacent vertices of m

// mstSet[v] is false for vertices not yet included in MST

// Update the key only if graph[u][v] is smaller than key[v]

if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])

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

}

// print the constructed MST

printMST(parent, V, graph);

}

// driver program to test above function

int main()

{

/\* Let us create the following graph

2 3

(0)--(1)--(2)

| / \ |

6| 8/ \5 |7

| / \ |

(3)-------(4)

9 \*/

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}};

// Print the solution

primMST(graph);

return 0;

}

**3. From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm4**

**Code:**

// A C++ program for Dijkstra's single source shortest path algorithm.

// The program is for adjacency matrix representation of the graph

#include <stdio.h>

#include <limits.h>

// Number of vertices in the graph

#define V 9

// A utility function to find the vertex with minimum distance value, from

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

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

{

// Initialize min value

int min = INT\_MAX, min\_index;

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

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

min = dist[v], min\_index = v;

return min\_index;

}

// A utility function to print the constructed distance array

int printSolution(int dist[], int n)

{

printf("Vertex Distance from Source\n");

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

printf("%d tt %d\n", i, dist[i]);

}

// Function that implements Dijkstra's single source shortest path algorithm

// for a graph represented using adjacency matrix representation

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

{

int dist[V]; // The output array. dist[i] will hold the shortest

// distance from src to i

bool sptSet[V]; // sptSet[i] will true if vertex i is included in shortest

// path tree or shortest distance from src to i is finalized

// Initialize all distances as INFINITE and stpSet[] as false

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

dist[i] = INT\_MAX, sptSet[i] = false;

// Distance of source vertex from itself is always 0

dist[src] = 0;

// Find shortest path for all vertices

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

{

// Pick the minimum distance vertex from the set of vertices not

// yet processed. u is always equal to src in the first iteration.

int u = minDistance(dist, sptSet);

// Mark the picked vertex as processed

sptSet[u] = true;

// Update dist value of the adjacent vertices of the picked vertex.

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

// Update dist[v] only if is not in sptSet, there is an edge from

// u to v, and total weight of path from src to v through u is

// smaller than current value of dist[v]

if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX

&& dist[u]+graph[u][v] < dist[v])

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

}

// print the constructed distance array

printSolution(dist, V);

}

// driver program to test above function

int main()

{

/\* Let us create the example graph discussed above \*/

int graph[V][V] = {{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}

};

dijkstra(graph, 0);

return 0;

}

1. **From a given vertex in a weighted connected graph, find shortest paths to other vertices negative weights (using Bellman-Ford algorithm).**

**Code:**

// A C++ program for Bellman-Ford's single source

// shortest path algorithm.

#include <bits/stdc++.h>

// a structure to represent a weighted edge in graph

struct Edge

{

int src, dest, weight;

};

// a structure to represent a connected, directed and

// weighted graph

struct Graph

{

// V-> Number of vertices, E-> Number of edges

int V, E;

// graph is represented as an array of edges.

struct Edge\* edge;

};

// Creates a graph with V vertices and E edges

struct Graph\* createGraph(int V, int E)

{

struct Graph\* graph = new Graph;

graph->V = V;

graph->E = E;

graph->edge = new Edge[E];

return graph;

}

// A utility function used to print the solution

void printArr(int dist[], int n)

{

printf("Vertex Distance from Source\n");

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

printf("%d \t\t %d\n", i, dist[i]);

}

// The main function that finds shortest distances from src to

// all other vertices using Bellman-Ford algorithm. The function

// also detects negative weight cycle

void BellmanFord(struct Graph\* graph, int src)

{

int V = graph->V;

int E = graph->E;

int dist[V];

// Step 1: Initialize distances from src to all other vertices

// as INFINITE

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

dist[i] = INT\_MAX;

dist[src] = 0;

// Step 2: Relax all edges |V| - 1 times. A simple shortest

// path from src to any other vertex can have at-most |V| - 1

// edges

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

{

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

{

int u = graph->edge[j].src;

int v = graph->edge[j].dest;

int weight = graph->edge[j].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

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

}

}

// Step 3: check for negative-weight cycles. The above step

// guarantees shortest distances if graph doesn't contain

// negative weight cycle. If we get a shorter path, then there

// is a cycle.

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

{

int u = graph->edge[i].src;

int v = graph->edge[i].dest;

int weight = graph->edge[i].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

printf("Graph contains negative weight cycle");

}

printArr(dist, V);

return;

}

// Driver program to test above functions

int main()

{

/\* Let us create the graph given in above example \*/

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

int E = 8; // Number of edges in graph

struct Graph\* graph = createGraph(V, E);

// add edge 0-1 (or A-B in above figure)

graph->edge[0].src = 0;

graph->edge[0].dest = 1;

graph->edge[0].weight = -1;

// add edge 0-2 (or A-C in above figure)

graph->edge[1].src = 0;

graph->edge[1].dest = 2;

graph->edge[1].weight = 4;

// add edge 1-2 (or B-C in above figure)

graph->edge[2].src = 1;

graph->edge[2].dest = 2;

graph->edge[2].weight = 3;

// add edge 1-3 (or B-D in above figure)

graph->edge[3].src = 1;

graph->edge[3].dest = 3;

graph->edge[3].weight = 2;

// add edge 1-4 (or A-E in above figure)

graph->edge[4].src = 1;

graph->edge[4].dest = 4;

graph->edge[4].weight = 2;

// add edge 3-2 (or D-C in above figure)

graph->edge[5].src = 3;

graph->edge[5].dest = 2;

graph->edge[5].weight = 5;

// add edge 3-1 (or D-B in above figure)

graph->edge[6].src = 3;

graph->edge[6].dest = 1;

graph->edge[6].weight = 1;

// add edge 4-3 (or E-D in above figure)

graph->edge[7].src = 4;

graph->edge[7].dest = 3;

graph->edge[7].weight = -3;

BellmanFord(graph, 0);

return 0;

}

1. **Print all the nodes reachable from a given starting node in a digraph using BFS method. Check whether a given graph is connected or not using DFS method.**

**Code: (BFS)**

// Program to print BFS traversal from a given

// source vertex. BFS(int s) traverses vertices

// reachable from s.

#include<iostream>

#include <list>

using namespace std;

// This class represents a directed graph using

// adjacency list representation

class Graph

{

int V; // No. of vertices

// Pointer to an array containing adjacency

// lists

list<int> \*adj;

public:

Graph(int V); // Constructor

// function to add an edge to graph

void addEdge(int v, int w);

// prints BFS traversal from a given source s

void BFS(int s);

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

void Graph::BFS(int s)

{

// Mark all the vertices as not visited

bool \*visited = new bool[V];

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

visited[i] = false;

// Create a queue for BFS

list<int> queue;

// Mark the current node as visited and enqueue it

visited[s] = true;

queue.push\_back(s);

// 'i' will be used to get all adjacent

// vertices of a vertex

list<int>::iterator i;

while(!queue.empty())

{

// Dequeue a vertex from queue and print it

s = queue.front();

cout << s << " ";

queue.pop\_front();

// Get all adjacent vertices of the dequeued

// vertex s. If a adjacent has not been visited,

// then mark it visited and enqueue it

for (i = adj[s].begin(); i != adj[s].end(); ++i)

{

if (!visited[\*i])

{

visited[\*i] = true;

queue.push\_back(\*i);

}

}

}

}

// Driver program to test methods of graph class

int main()

{

// Create a graph given in the above diagram

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Breadth First Traversal "

<< "(starting from vertex 2) \n";

g.BFS(2);

return 0;

}

**Code: (DFS)**

// C++ program to check if a given directed graph is strongly

// connected or not

#include <iostream>

#include <list>

#include <stack>

using namespace std;

class Graph

{

int V; // No. of vertices

list<int> \*adj; // An array of adjacency lists

// A recursive function to print DFS starting from v

void DFSUtil(int v, bool visited[]);

public:

// Constructor and Destructor

Graph(int V) { this->V = V; adj = new list<int>[V];}

~Graph() { delete [] adj; }

// Method to add an edge

void addEdge(int v, int w);

// The main function that returns true if the graph is strongly

// connected, otherwise false

bool isSC();

// Function that returns reverse (or transpose) of this graph

Graph getTranspose();

};

// A recursive function to print DFS starting from v

void Graph::DFSUtil(int v, bool visited[])

{

// Mark the current node as visited and print it

visited[v] = true;

// Recur for all the vertices adjacent to this vertex

list<int>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

if (!visited[\*i])

DFSUtil(\*i, visited);

}

// Function that returns reverse (or transpose) of this graph

Graph Graph::getTranspose()

{

Graph g(V);

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

{

// Recur for all the vertices adjacent to this vertex

list<int>::iterator i;

for(i = adj[v].begin(); i != adj[v].end(); ++i)

{

g.adj[\*i].push\_back(v);

}

}

return g;

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

// The main function that returns true if graph is strongly connected

bool Graph::isSC()

{

// St1p 1: Mark all the vertices as not visited (For first DFS)

bool visited[V];

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

visited[i] = false;

// Step 2: Do DFS traversal starting from first vertex.

DFSUtil(0, visited);

// If DFS traversal doesn’t visit all vertices, then return false.

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

if (visited[i] == false)

return false;

// Step 3: Create a reversed graph

Graph gr = getTranspose();

// Step 4: Mark all the vertices as not visited (For second DFS)

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

visited[i] = false;

// Step 5: Do DFS for reversed graph starting from first vertex.

// Staring Vertex must be same starting point of first DFS

gr.DFSUtil(0, visited);

// If all vertices are not visited in second DFS, then

// return false

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

if (visited[i] == false)

return false;

return true;

}

// Driver program to test above functions

int main()

{

// Create graphs given in the above diagrams

Graph g1(5);

g1.addEdge(0, 1);

g1.addEdge(1, 2);

g1.addEdge(2, 3);

g1.addEdge(3, 0);

g1.addEdge(2, 4);

g1.addEdge(4, 2);

g1.isSC()? cout << "Yes\n" : cout << "No\n";

Graph g2(4);

g2.addEdge(0, 1);

g2.addEdge(1, 2);

g2.addEdge(2, 3);

g2.isSC()? cout << "Yes\n" : cout << "No\n";

return 0;

}