Discrete Structure Practical Assignment

Submitted to – Shalini Ma’am Submitted by – Suyash Pratap Mall Roll no. - 88051

# Write a Program to represent Graphs using the Adjacency Matrices and check if it is a complete graph.

ANS -

#include<iostream> using namespace std;

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

//for displaying the entered matrix 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;

}

}

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

vertArr[u][v] = 1;

vertArr[v][u] = 1;

}

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

int v,flag=0,flag2=0; int rolt=0,rol=0,col=0; int edgeno;

cout<<"Enter the no. of vertices of the Graph : "; cin>>v;

cout<<"Enter 1 if there is edge between given vertices else enter 0."<<endl;

while(flag<v\*v)

{

if(rol!=col && vertArr[rol][col]!=1){//for eliminating self loop

cout<<"( "<<rol<<","<<col<<" ) : ";

cin>>flag2;

}

if(flag2==1)

{

add\_edge(rol,col);

edgeno++;

}

flag++;

flag2=0;//making edge 0 zero

col++;

rolt++;

if(rolt==v)//for incrementing row

{ rol++;

rolt=0;

}

if(col==v)

{col=0;}

}

cout<<endl<<"Entered Graph looks like :"<<endl; displayMatrix(v);

cout<<endl;

//to check whether it is a complete graph or not by formula if(edgeno==((v\*(v-1))/2))

{

cout<<"Entered graph is a complete Graph";

}

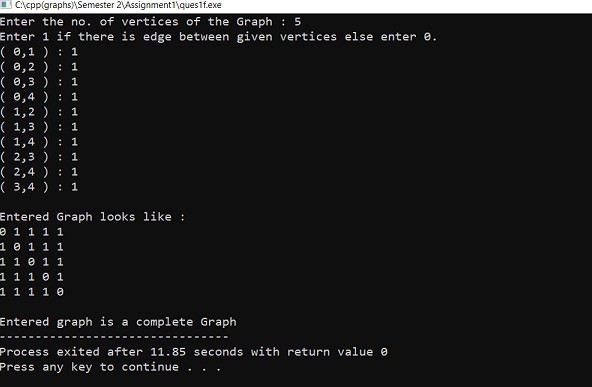
else{

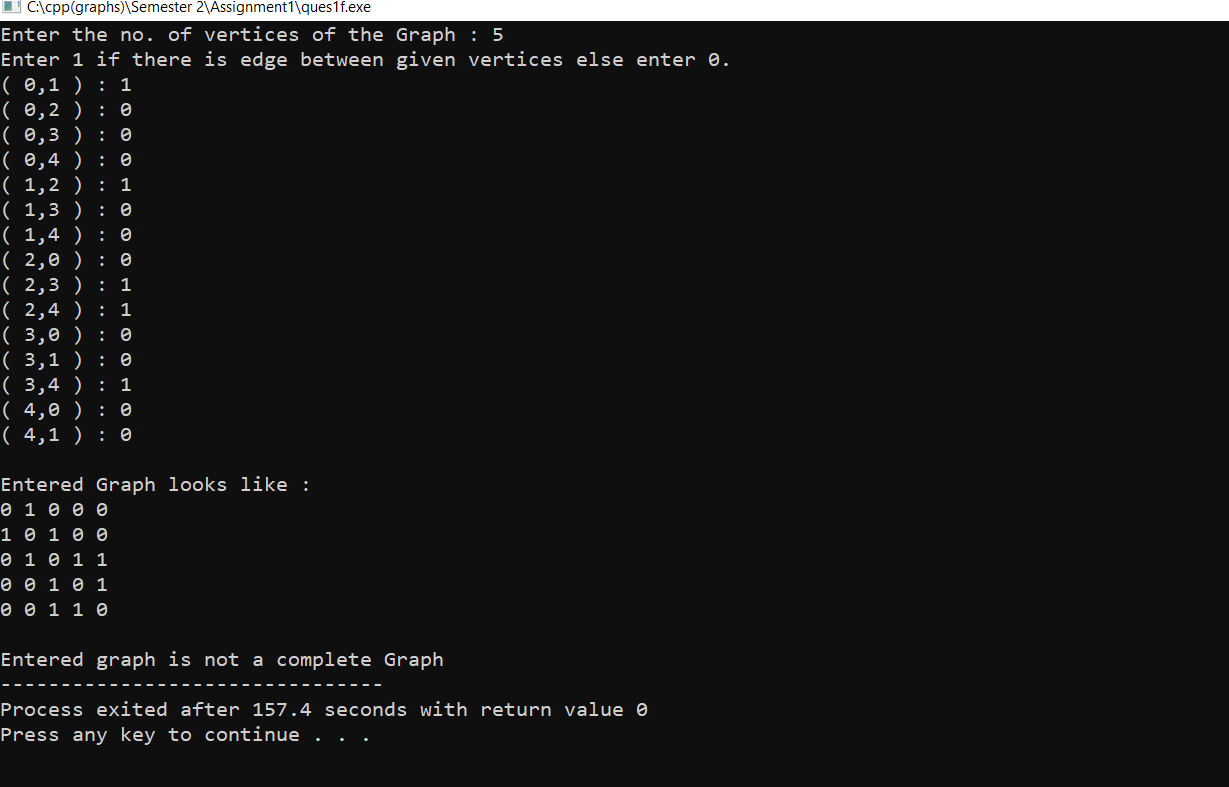
cout<<"Entered graph is not a complete Graph";

}

}

Output(s):





# Write a Program to accept a directed graph G and compute the in-degree and out-degree of each vertex. ANS -

#include<iostream> using namespace std;

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

int in[20]; int out[20];

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;

out[u]+=1;

}

main(int argc, char\* argv[]) { int v,flag=0,flag2=0;

int rolt=0,rol=0,col=0; int no=0;

cout<<"Enter the no. of vertices of the Graph : "; cin>>v;

cout<<"Enter 1 if there is edge between given vertexses else enter 0."<<endl; cout<<"(head , tail)"<<endl;

while(flag<v\*v)

{

if(rol!=col)

{

cout<<"( "<<rol<<","<<col<<" ) : ";

{

add\_edge(rol,col);

no++;

}

flag++;

flag2=0;

if(rolt==v)

{ rol++;

rolt=0;

}

if(col==v)

}

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

{

cout<<endl;

cout<<"In-Degree of "<<i<<" = "<<in[i]<<endl;

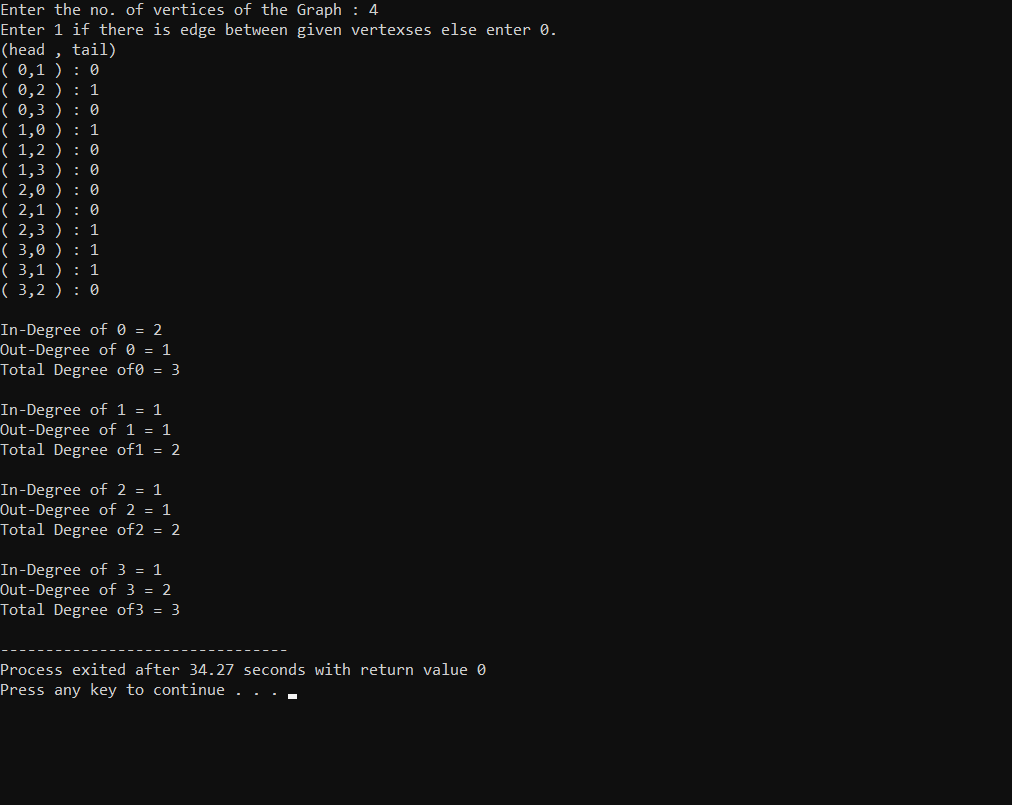
cout<<"Out-Degree of "<<i<<" = "<<out[i]<<endl;

cout<<"Total Degree of"<<i<<" = "<<in[i]+out[i];

}

}

Output:



# Given a graph G, Write a Program to find the number of paths of length n between the source and destination entered by the user.

ANS-

#include<iostream> #include<vector> #include<list>

using namespace std; vector<vector<int> > v;

// Graph 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;

// A recursive function used by printAllPaths()

void printAllPathsUtil(int v1, int v2, bool visited[], int path[], int index); public:

Graph(int V); // Constructor

bool pathExist; //variable to indicate if path exist or not

// function to add an edge to graph

void addEdge(int v, int w);

void printAllPaths(int v1, int v2);

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

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

{

}

void Graph::printAllPathsUtil(int v1, int v2, bool visited[], int path[], int index)

{

// Mark the current node as visited and store it in path[]

visited[v1] = true;

path[index]=v1;

index++;

// If current vertex is same as v2, then print

// current path[]

if(v1==v2){

int i;

if(!pathExist)

//cout<<"Following are the paths between "<<path[0]<<" and "<<path[index-1]<<endl;

pathExist=true;

vector<int> temp;

for(i=0;i<index-1;i++)

//cout<<path[i]<<"->";

temp.push\_back(path[i]); }

//cout<<path[i]<<endl;

temp.push\_back(path[i]);

v.push\_back(temp);

}

else{ // If current vertex is not v2

// Recur for all the vertices adjacent

// to this vertex

list<int>::iterator i;

for (i = adj[v1].begin(); i != adj[v1].end(); ++i)

if (!visited[\*i])

printAllPathsUtil(\*i, v2, visited, path, index);

}

// Remove current vertex from path[] and mark it as unvisited

index--;

visited[v1]=false;

}

void Graph::printAllPaths(int v1, int v2)

{

// Mark all the vertices as not visited

bool \*visited = new bool[V];

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

visited[i] = false;

// Create an array to store paths

int \*path = new int[V];

// Initialize path[] as empty

int index = 0;

pathExist=false;

// Call the recursive helper function to print all paths

printAllPathsUtil(v1,v2,visited,path,index);

}

// Driver code int main()

{

// Create a graph given in the above diagram

Graph g(8);

g.addEdge(0,1);

g.addEdge(1, 3);

g.addEdge(6, 4);

int v1=0, v2=6;

int a,b,len;

cout<<"Enter the source : ";

cin>>a;

cout<<"Enter the destination : ";

cin>>b;

cout<<"Enter the length n : ";

cin>>len;

g.printAllPaths(a,b);

// if(!g.pathExist){

// cout<<"There is no path exist between "<<v1<<" and "<<v2;

// }

int count=0;

for(int i=0;i<v.size();i++)

{

vector<int > temp;

temp=v[i];

if(temp.size()==len)

count++;

}

cout<<endl; if(count)

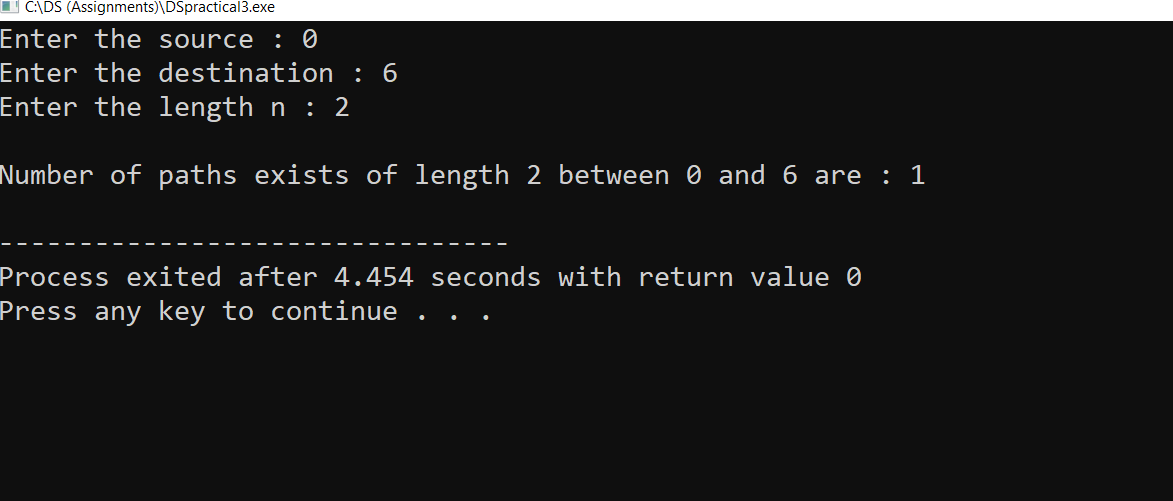
cout<<"No. of paths exists of length "<<len<<" between "<<a<<" and "<<b<<" are : "<<count<<endl;

else

cout<<"No such path exists!"<<endl;

}

# OUTPUT:



1. Given an adjacency matrix of a graph, write a program to check whether a given set of vertices {v1,v2,v3.....,vk} forms an Euler path / Euler Circuit (for circuit assume vk=v1). ANS -

// A C++ program to check if a given graph is Eulerian or not #include<iostream>

#include <list>

using namespace std;

// A class that represents an undirected graph class Graph

{

int V; // No. of vertices

list<int> \*adj; // A dynamic array of adjacency lists public:

// Constructor and destructor

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

~Graph() { delete [] adj; } // To avoid memory leak

// function to add an edge to graph void addEdge(int v, int w);

// Method to check if this graph is Eulerian or not int isEulerian();

// Method to check if all non-zero degree vertices are connected

bool isConnected();

// Function to do DFS starting from v. Used in isConnected(); void DFSUtil(int v, bool visited[]);

};

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

{

adj[v].push\_back(w);

adj[w].push\_back(v); // Note: the graph is undirected

}

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);

}

// Method to check if all non-zero degree vertices are connected.

// It mainly does DFS traversal starting from bool Graph::isConnected()

{

// Mark all the vertices as not visited bool visited[V];

int i;

for (i = 0; i < V; i++) visited[i] = false;

// Find a vertex with non-zero degree for (i = 0; i < V; i++)

if (adj[i].size() != 0) break;

// If there are no edges in the graph, return true if (i == V)

return true;

// Start DFS traversal from a vertex with non-zero degree DFSUtil(i, visited);

// Check if all non-zero degree vertices are visited for (i = 0; i < V; i++)

if (visited[i] == false && adj[i].size() > 0) return false;

return true;

}

/\* The function returns one of the following values 0 --> If grpah is not Eulerian

1 --> If graph has an Euler path (Semi-Eulerian) 2 --> If graph has an Euler Circuit (Eulerian) \*/ int Graph::isEulerian()

{

// Check if all non-zero degree vertices are connected if (isConnected() == false)

return 0;

// Count vertices with odd degree int odd = 0;

for (int i = 0; i < V; i++) if (adj[i].size() & 1)

odd++;

// If count is more than 2, then graph is not Eulerian if (odd > 2)

return 0;

// If odd count is 2, then semi-eulerian.

// If odd count is 0, then eulerian

// Note that odd count can never be 1 for undirected graph return (odd)? 1 : 2;

}

// Function to run test cases void test(Graph &g)

{

int res = g.isEulerian(); if (res == 0)

cout << "graph is not Eulerian\n"; else if (res == 1)

cout << "graph has a Euler path\n";

else

}

cout << "graph has a Euler cycle\n";

// Driver program to test above function int main()

{

// Graph with 5 edges int n, edge;

cout<<"Enter the number of vertex: "<<endl; cin >> n;

cout<<"Enter the number of edges: "<<endl; cin>>edge;

Graph g1(n);

for (int i = 0; i<edge; i++) { int a, b, counter; counter = i+1 ;

cout<<"Enter the vextex pair connected through edges: "<< counter<<endl;

cin>>a>>b; g1.addEdge(a, b);

;

cout<<"Number of edges left: "<<edge - counter<< endl;

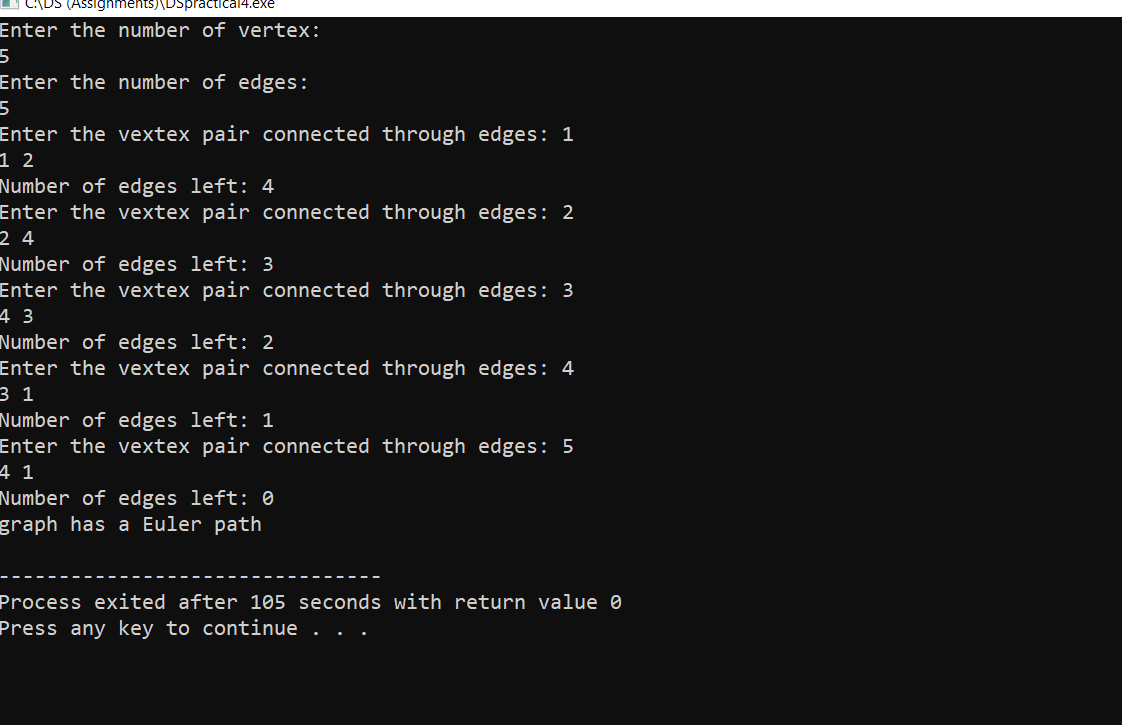
}

test(g1);

return 0;

}

OUTPUT:



1. Given a full m-ary tree with i internal vertices, Write a Program to find the number of leaf nodes.

AND-

#include<iostream> using namespace std;

//returns the number of leafs (no. of leafs = internal\_vertices \* (m-1) + 1) int calLeafNod(int m, int i)

{

int leaf = 0;

leaf = i \* (m-1) + 1;

return leaf;

}

int main()

{

int m,i;

cout<<"Enter the value of m in an m-ary Tree , m = ";

cin>>m;

cout<<"Enter the number of Internal Vertices : ";

cin>>i;

cout<<"Total number of Leaf Nodes are : "<<calLeafNod(m,i);

}

OUTPUT:

