**CHANDIGARH UNIVERSITY**

**UNIVERSITY INSTITUTE OF ENGINEERING**

**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**



|  |  |
| --- | --- |
| **Submitted By: Kanishk Soni Submitted To: Er. Tanu Dhiman** | |
| **Subject Name** | **Design and Analysis of Algorithm Lab** |
| **Subject Code** | **20CSP-312** |
| **Branch** | **BE-CSE** |
| **Semester** | **5th** |

**Worksheet - 8**

**Student Name:** Kanishk Soni **UID:** 20BCS9398

**Branch:** BE CSE **Section/Group:** 20BCS\_WM\_707-B

**Semester:** 5 **Date of Performance:** 09/11/2022

**Subject Name:** Design & Analysis of Algorithm **Subject Code:** 20CSP-312

1. **Aim/Overview of the practical:** Code and analyze to do a depth-first search (DFS) on an undirected graph. Implementing an application of DFS such as (i) to find the topological sort of a directed acyclic graph, OR (ii) to find a path from source to goal in a maze.
2. **Program code:**
3. Code and analyze to do a depth-first search (DFS) on an undirected graph

#include <iostream>

#include <map>

#include <list>

using namespace std;

class Graph {

public:

map<int, bool> visited;

map<int, list<int> > adj;

void addEdge(int v, int w);

void DFS(int v);

};

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

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

}

void Graph::DFS(int v) {

visited[v] = true;

cout << v << " ";

list<int>::iterator i;

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

if (!visited[\*i])

DFS(\*i);

}

int main() {

Graph g;

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 Depth First Traversal (starting from vertex 2) \n";

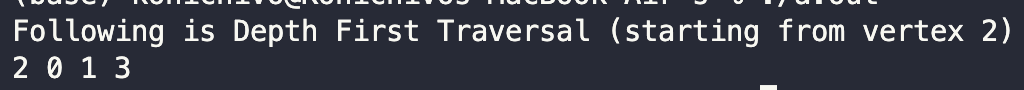
g.DFS(2);

cout<<endl;

return 0;

}

**Output:**



1. to find the topological sort of a directed acyclic graph

#include <iostream>

#include <list>

#include <stack>

using namespace std;

class Graph {

int V;

list<int> \*adj;

void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);

public:

Graph(int V);

void addEdge(int v, int w);

void topologicalSort();

};

Graph::Graph(int V) {

this->V = V;

adj = new list<int>[V];

}

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

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

}

void Graph::topologicalSortUtil(int v, bool visited[], stack<int> &Stack) {

visited[v] = true;

list<int>::iterator i;

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

if (!visited[\*i])

topologicalSortUtil(\*i, visited, Stack);

Stack.push(v);

}

void Graph::topologicalSort() {

stack<int> Stack;

bool \*visited = new bool[V];

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

visited[i] = false;

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

if (visited[i] == false)

topologicalSortUtil(i, visited, Stack);

while (Stack.empty() == false)

{

cout << Stack.top() << " ";

Stack.pop();

}

}

int main() {

Graph g(6);

g.addEdge(5, 2);

g.addEdge(5, 0);

g.addEdge(4, 0);

g.addEdge(4, 1);

g.addEdge(2, 3);

g.addEdge(3, 1);

cout << "Following is a Topological Sort of the given graph \n";

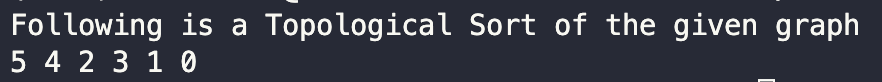
g.topologicalSort();

cout<<endl;

return 0;

}

**Output:**



1. to find a path from source to goal in a maze.

#include <iostream>

#include <vector>

#include <climits>

#include <cstring>

using namespace std;

bool isSafe(vector<vector<int> > &mat, vector<vector<bool> > &visited, int x, int y) {

return (x >= 0 && x < mat.size() && y >= 0 && y < mat[0].size()) && mat[x][y] == 1 && !visited[x][y];

}

void findShortestPath(vector<vector<int> > &mat, vector<vector<bool> > &visited, int i, int j, int x, int y, int &min\_dist, int dist) {

if (i == x && j == y) {

min\_dist = min(dist, min\_dist);

return;

}

visited[i][j] = true;

if (isSafe(mat, visited, i + 1, j)) {

findShortestPath(mat, visited, i + 1, j, x, y, min\_dist, dist + 1);

}

if (isSafe(mat, visited, i, j + 1)) {

findShortestPath(mat, visited, i, j + 1, x, y, min\_dist, dist + 1);

}

if (isSafe(mat, visited, i - 1, j)) {

findShortestPath(mat, visited, i - 1, j, x, y, min\_dist, dist + 1);

}

if (isSafe(mat, visited, i, j - 1)) {

findShortestPath(mat, visited, i, j - 1, x, y, min\_dist, dist + 1);

}

visited[i][j] = false;

}

int findShortestPathLength(vector<vector<int> > &mat, pair<int, int> &src, pair<int, int> &dest) {

if (mat.size() == 0 || mat[src.first][src.second] == 0 || mat[dest.first][dest.second] == 0) {

return -1;

}

int M = mat.size();

int N = mat[0].size();

vector<vector<bool> > visited;

visited.resize(M, vector<bool>(N));

int min\_dist = INT\_MAX;

findShortestPath(mat, visited, src.first, src.second, dest.first, dest.second, min\_dist, 0);

if (min\_dist != INT\_MAX) {

return min\_dist;

}

return -1;

}

int main() {

vector<vector<int> > mat = {

{1, 1, 1, 1, 1, 0, 0, 1, 1, 1},

{0, 1, 1, 1, 1, 1, 0, 1, 0, 1},

{0, 0, 1, 0, 1, 1, 1, 0, 0, 1},

{1, 0, 1, 1, 1, 0, 1, 1, 0, 1},

{0, 0, 0, 1, 0, 0, 0, 1, 0, 1},

{1, 0, 1, 1, 1, 0, 0, 1, 1, 0},

{0, 0, 0, 0, 1, 0, 0, 1, 0, 1},

{0, 1, 1, 1, 1, 1, 1, 1, 0, 0},

{1, 1, 1, 1, 1, 0, 0, 1, 1, 1},

{0, 0, 1, 0, 0, 1, 1, 0, 0, 1},

};

pair<int, int> src = make\_pair(0, 0);

pair<int, int> dest = make\_pair(7, 5);

int min\_dist = findShortestPathLength(mat, src, dest);

if (min\_dist != -1) {

cout << "The shortest path from source to destination has length "<< min\_dist;

} else {

cout << "Destination cannot be reached from a given source";

}

cout<<endl;

return 0;

}

**Output:**

