**DFS (not work on dev)**

#include <iostream>

#include <vector>

#include <list>

using namespace std;

class Graph {

int V; // Number of vertices

list<int> \*adj; // Pointer to an array containing adjacency lists

void DFSUtil(int v, vector<bool> &visited, int &count); // Utility function for DFS

public:

Graph(int V); // Constructor

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

void DFS(int v); // Function to perform DFS traversal from a given vertex

};

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::DFSUtil(int v, vector<bool> &visited, int &count) {

// Mark the current node as visited and print it

visited[v] = true;

count++; // Counting the operation

cout << v << " ";

// Recur for all the vertices adjacent to this vertex

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

if (!visited[\*i]) {

DFSUtil(\*i, visited, count);

}

count++; // Counting the operation

}

}

void Graph::DFS(int v) {

vector<bool> visited(V, false); // Mark all the vertices as not visited

int count = 0; // Initialize the operation count

// Call the recursive helper function to print DFS traversal

DFSUtil(v, visited, count);

cout << "\nNumber of operations: " << count << endl;

}

int main() {

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

g.DFS(2);

return 0;

}

**BFS(not work on DEV)**

#include <iostream>

#include <vector>

#include <list>

#include <queue>

using namespace std;

class Graph {

int V; // Number of vertices

list<int> \*adj; // Pointer to an array containing adjacency lists

public:

Graph(int V); // Constructor

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

void BFS(int s); // Function to perform BFS traversal from a given source

};

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) {

vector<bool> visited(V, false); // Mark all the vertices as not visited

queue<int> q; // Create a queue for BFS

int count = 0; // Initialize the operation count

// Mark the current node as visited and enqueue it

visited[s] = true;

q.push(s);

count++; // Counting the operation

while (!q.empty()) {

// Dequeue a vertex from queue and print it

s = q.front();

cout << s << " ";

q.pop();

count++; // Counting the operation

// Get all adjacent vertices of the dequeued vertex s

// If an adjacent has not been visited, mark it visited and enqueue it

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

if (!visited[\*i]) {

visited[\*i] = true;

q.push(\*i);

count++; // Counting the operation

}

}

}

cout << "\nNumber of operations: " << count << endl;

}

int main() {

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 << "Breadth First Traversal (starting from vertex 2):\n";

g.BFS(2);

return 0;

}

**HAMILTONIAN CYCLE**

#include <iostream>

#include <vector>

using namespace std;

class Graph {

int V; // Number of vertices

vector<vector<int> > adj; // Adjacency matrix

public:

Graph(int V); // Constructor

void addEdge(int u, int v); // Function to add an edge to the graph

bool hamiltonianCycle(); // Function to find Hamiltonian Cycle

bool hamiltonianCycleUtil(vector<int>& path, vector<bool>& visited, int pos); // Utility function for finding Hamiltonian Cycle

};

Graph::Graph(int V) {

this->V = V;

adj.resize(V, vector<int>(V, 0));

}

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

adj[u][v] = 1;

adj[v][u] = 1; // Undirected graph

}

bool Graph::hamiltonianCycle() {

vector<int> path(V, -1); // Path to store the Hamiltonian Cycle

vector<bool> visited(V, false); // Mark all vertices as not visited

// Start from vertex 0 (assuming 0 based indexing)

path[0] = 0;

visited[0] = true;

if (hamiltonianCycleUtil(path, visited, 1)) {

// Print the Hamiltonian Cycle found

cout << "Hamiltonian Cycle exists: ";

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

cout << path[i] << " ";

}

cout << path[0] << endl;

return true;

} else {

cout << "Hamiltonian Cycle does not exist" << endl;

return false;

}

}

bool Graph::hamiltonianCycleUtil(vector<int>& path, vector<bool>& visited, int pos) {

if (pos == V) {

// Check if there is an edge from the last added vertex to the first vertex

if (adj[path[pos - 1]][path[0]] == 1) {

return true;

} else {

return false;

}

}

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

// Check if this vertex can be added to the Hamiltonian Cycle

if (!visited[v] && adj[path[pos - 1]][v] == 1) {

path[pos] = v;

visited[v] = true;

if (hamiltonianCycleUtil(path, visited, pos + 1)) {

return true;

}

// Backtrack

path[pos] = -1;

visited[v] = false;

}

}

return false;

}

int main() {

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

Graph g(V);

// Adding edges to the graph to form a cycle

g.addEdge(0, 1);

g.addEdge(1, 2);

g.addEdge(2, 3);

g.addEdge(3, 4);

g.addEdge(4, 0);

cout << "Finding Hamiltonian Cycle in the graph:\n";

g.hamiltonianCycle();

return 0;

}