**Experiment No 4**

**AIM:** Study of Breadth First Traversal and Depth First Traversal for a user defined Graph.

**PROBLEM STATEMENT:** Write a C++ Program using Breadth First Traversal and Depth First Traversal for a user defined Graph.

**REQUIREMENT:**Turbo C/ GCC Compiler

**OPERATING SYSTEM:** Windows/Linux/Unix.

**THEORY:**

**Breadth First Search (BFS):**

BFS is a fundamental graph traversal algorithm. It begins with a node, then first traverses all its adjacent nodes. Once all adjacent are visited, then their adjacent are traversed.

**BFS from a Given Source:**

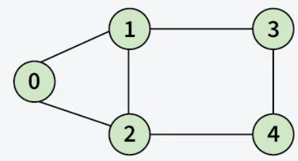
The algorithm starts from a given source and explores all reachable vertices from the given source. It is similar to the [Breadth-First Traversal of a tree](https://www.geeksforgeeks.org/level-order-tree-traversal/). Like tree, we begin with the given source (in tree, we begin with root) and traverse vertices level by level using a queue data structure. The only catch here is that, unlike trees, graphsmay contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a **boolean** **visited** array.

**Initialization:**Enqueue the given source vertex into a queue and mark it as visited.

1. **Exploration:**While the queue is not empty:
   * Dequeue a node from the queue and visit it (e.g., print its value).
   * For each unvisited neighbor of the dequeued node:
     + Enqueue the neighbor into the queue.
     + Mark the neighbor as visited.
2. **Termination:**Repeat step 2 until the queue is empty.

This algorithm ensures that all nodes in the graph are visited in a breadth-first manner, starting from the starting node.

**Example:**



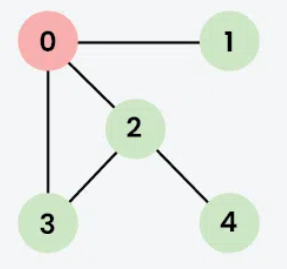
|  |  |
| --- | --- |
| **Step 1: Initially Queue and array is empty** |  |
| **Step 2:Push 0 into queue and marked as visible** |  |
| **Step 3: Remove 0 from front of queue and visit unvisited neighbors (1, 2) and insert in to queue** |  |
| **Step 4: Remove 1 from front of queue and visit unvisited neighbor (3) and insert in to queue** |  |
| **Step 5: Remove 2 from front of queue and visit unvisited neighbor (4) and insert in to queue** |  |
| **Step 6: Remove 3 from front of queue and there is no unvisited neighbors** |  |
| **Step 7: Remove 4 from front of queue and there is no unvisited neighbors** |  |
| **Step 8: STOP because Queue is empty** |  |

**Depth First Search (DF):**

In Depth First Search (or DFS) for a graph, we traverse all adjacent vertices one by one. When we traverse an adjacent vertex, we completely finish the traversal of all vertices reachable through that adjacent vertex. This is similar to a tree, where we first completely traverse the left subtree and then move to the right subtree. The key difference is that, unlike trees, graphs may contain cycles (a node may be visited more than once). To avoid processing a node multiple times, we use a boolean visited array.

For DFS we used stack as data structure.

**Example:**



|  |  |  |
| --- | --- | --- |
| **Step 1: Initially stack and array is empty** |  |  |
| **Step 2: Visit 0 and push non-visited adjacent nodes on stack** |  |  |
| **Step 3: pop 1 element from stack (1) push non-visited adjacent nodes on stack** |  |  |
| **Step 4: pop 1 element from stack (2) push non-visited adjacent nodes (4) on stack** |  |  |
| **Step 5: pop 1 element from stack (4) push non-visited adjacent nodes on stack** |  |  |
| **Step 6: pop 1 element from stack (3) push non-visited adjacent nodes on stack** |  |  |

**CONCLUSION:** Program for BFS and DFS is implemented successfully.

// C++ program for BFS and DFS of an undirected graph

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

// BFS from given source s

void bfs(vector<vector<int>>& adj, int s)

{

// Create a queue for BFS

queue<int> q;

vector<bool> visited(adj.size(), false);

visited[s] = true;

q.push(s);

while (!q.empty()) {

int curr = q.front();

q.pop();

cout << curr << " ";

for (int x : adj[curr]) {

if (!visited[x]) {

visited[x] = true;

q.push(x);

}

}

}

}

// Function to add an edge to the graph BFS

void addEdge(vector<vector<int>>& adj,

int u, int v)

{

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

adj[v].push\_back(u); // Undirected Graph

}

// Recursive function for DFS traversal

void DFSRec(vector<vector<int>> &adj, vector<bool> &visited, int s){

visited[s] = true;

// Print the current vertex

cout << s << " ";

// Recursively visit all adjacent vertices

// that are not visited yet

for (int i : adj[s])

if (visited[i] == false)

DFSRec(adj, visited, i);

}

void DFS(vector<vector<int>> &adj, int s){

vector<bool> visited(adj.size(), false);

DFSRec(adj, visited, s);

}

// To add an edge in an undirected graph DFS

void addEdge1(vector<vector<int>> &adj, int s, int t){

adj[s].push\_back(t);

adj[t].push\_back(s);

}

int main(){

int V = 5;

int ch;

int s = 1; // Starting vertex for DFS

vector<vector<int>> adj(V);

vector<vector<int>> edges={{1, 2},{1, 0},{2, 0},{2, 3},{2, 4}};

while(1)

{

cout<<"\nEnter 1: BFS 2: DFS 3: Exit";

cin>>ch;

switch(ch)

{

case 1:

// Add edges to the graph

addEdge(adj, 0, 1);

addEdge(adj, 0, 2);

addEdge(adj, 1, 3);

addEdge(adj, 1, 4);

addEdge(adj, 2, 4);

cout << "BFS starting from 0 : \n";

bfs(adj, 0);

break;

case 2:

for (auto &e : edges)

addEdge1(adj, e[0], e[1]);

cout << "DFS from source: " << s << endl;

DFS(adj, s); // Perform DFS starting from the source vertex

break;

case 3:

return 0;

}

}

return 0;

}