Assignment: 03

**Course title:** Data Structure Laboratory

**Course Code:** CSE 212

**Date Of Submission:** 25-04-2025

**Submitted to,**

**Name:** Mohammad Akbar Bin Shah

**Designation:** Lecturer, SoSET

**Submitted by,**

**Name:** Mohammad Fahim

**ID:** 242002112

**Section:** 6

**Department:** CSE

**Graph Traversal:**

**Code:**

#include <iostream>

#include <vector>

#include <queue>

#include <limits>

using namespace std;

struct graph {

int vertices;

vector<vector<int>> adjmatrix;

vector<vector<int>> adjlist;

graph(int v) {

vertices = v;

adjmatrix.resize(v, vector<int>(v, 0));

adjlist.resize(v);

}

void addedge(int u, int v) {

adjmatrix[u][v] = 1;

adjmatrix[v][u] = 1;

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

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

}

void displayadjmatrix() {

cout << "\nAdjacency Matrix:\n";

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

for (int j = 0; j < vertices; ++j) {

cout << adjmatrix[i][j] << " ";

}

cout << endl;

}

}

void displayadjlist() {

cout << "\nAdjacency List:\n";

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

cout << i << ": ";

for (int neighbor : adjlist[i]) {

cout << neighbor << " ";

}

cout << endl;

}

}

void bfs(int start) {

vector<bool> visited(vertices, false);

queue<int> q;

cout << "\nBFS Traversal starting from vertex " << start << ": ";

visited[start] = true;

q.push(start);

while (!q.empty()) {

int current = q.front();

q.pop();

cout << current << " ";

for (int neighbor : adjlist[current]) {

if (!visited[neighbor]) {

visited[neighbor] = true;

q.push(neighbor);

}

}

}

cout << endl;

}

};

graph\* buildgraph() {

int v, e;

cout << "Enter number of vertices: ";

cin >> v;

while (v <= 0) {

cout << "Number of vertices must be greater than 0. Please try again: ";

cin >> v;

}

cout << "Enter number of edges: ";

cin >> e;

while (e < 0 || e > (v \* (v - 1)) / 2) {

cout << "Invalid number of edges! Enter a number between 0 and " << (v \* (v - 1)) / 2 << ": ";

cin >> e;

}

graph\* g = new graph(v);

cout << "Enter edges (u v) format where 0 <= u, v < " << v << ":\n";

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

int u, v;

cout << "Edge " << (i + 1) << ": ";

cin >> u >> v;

if (u >= 0 && u < g->vertices && v >= 0 && v < g->vertices && u != v) {

g->addedge(u, v);

} else {

cout << "Invalid edge! Vertices must be between 0 and " << g->vertices - 1 << " and u != v. Please enter again.\n";

--i;

}

}

return g;

}

int main() {

graph\* g = buildgraph();

g->displayadjmatrix();

g->displayadjlist();

int start;

cout << "\nEnter starting vertex for BFS: ";

cin >> start;

if (start >= 0 && start < g->vertices) {

g->bfs(start);

} else {

cout << "Invalid starting vertex! Vertex must be between 0 and " << g->vertices - 1 << ".\n";

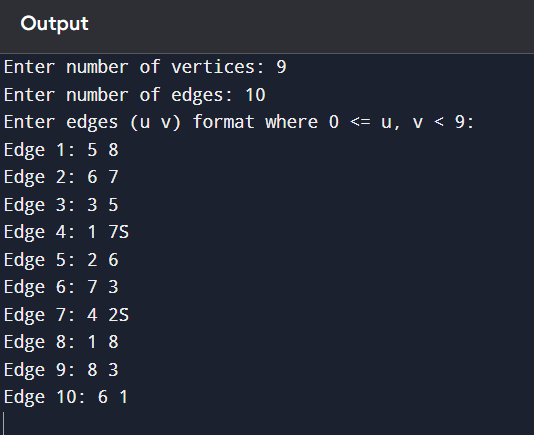
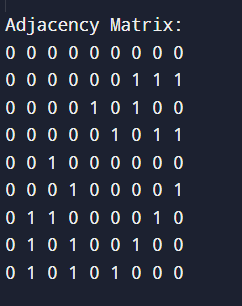
}

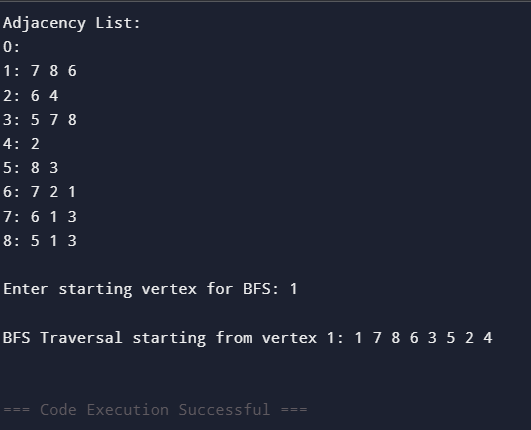
delete g;

return 0;

}

**Input/Output:**

****

****

**Explanation:**

The approach solves the problem by using an undirected graph represented by both an adjacency matrix and an adjacency list.

* **Graph Construction**: The program allows the user to input the number of vertices and edges. It validates the input, ensuring the number of vertices is positive and the number of edges is within valid bounds. The edges are then added to both the adjacency matrix and the adjacency list, maintaining the undirected graph structure.
* **Edge Addition**: The addedge function adds edges to both the adjacency matrix (by setting adjmatrix[u][v] and adjmatrix[v][u] to 1) and the adjacency list (by adding v to adjlist[u] and u to adjlist[v]).
* **BFS Traversal**: The program performs a Breadth-First Search (BFS) traversal starting from a user-defined vertex. The bfs function uses a queue and a visited array to explore the graph level by level, printing the vertices in the order they are visited.
* **Graph Display**: The displayadjmatrix and displayadjlist functions display the graph in two different formats—matrix and list—allowing the user to visualize the graph structure.
* **User Input**: The buildgraph function takes user input to construct the graph dynamically, ensuring that the graph’s properties remain valid throughout.

This approach provides a comprehensive solution to graph construction, traversal, and visualization while ensuring the graph structure is consistent and valid.