//Assignment no 6

#include <iostream>

using namespace std;

class graph

{

public:

int vcnt,rcnt;

int a[20][20];

int v[20];

graph()

{

vcnt=0;

rcnt=0;

}

void admat();

void dis\_admat();

void bfs();

void dfs();

int search(int key);

};

void graph::admat()

{

cout<<"Enter total no. of vertices:";

cin>>vcnt;

for(int i=1;i<=vcnt;i++)

{

for(int j=1;j<=vcnt;j++)

{

cout<<"Edge from "<<i<<" to "<<j<<":";

cin>>a[i][j];

}

}

}

void graph::dis\_admat()

{

for(int i=1;i<=vcnt;i++)

{

for(int j=1;j<=vcnt;j++)

{

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

}

cout<<endl;

}

}

class queue

{

public:

int q[20];

int front,rear;

queue()

{

front=0;

rear=0;

}

void insert(int key);

int del();

};

void queue::insert(int key)

{

if(rear<20)

{

q[rear++]=key;

}

else

{

cout<<"Queue is full."<<endl;

}

}

int queue::del()

{

if(front!=rear)

{

return(q[front++]);

}

else

{

cout<<"Queue is empty."<<endl;\

return -1;

}

}

class stack

{

public:

int s[20];

int top;

stack()

{

top=-1;

}

void push(int key);

int pop();

};

void stack::push(int key)

{

if(top<=20)

{

s[++top]=key;

}

else

{

cout<<"Stack is full."<<endl;

}

}

int stack::pop()

{

if(top!=-1)

{

return s[top--];

}

else

{

cout<<"Stack is empty."<<endl;

return -1;

}

}

int graph::search(int key)

{

//search val in visited array. if val is already present return 1 else return 0

int i;

for(i=0;i<rcnt;i++)

{

if(v[i]==key)

{

return (1);

}

}

if(i==rcnt)

{

return (0);

}

return -1;

}

void graph::bfs()

{

int sv,curr; //sv= start vertex

cout<<"Enter start vertex:";

cin>>sv;

v[rcnt++]=sv;

queue q;

q.insert(sv);

for(int i=1;i<=vcnt;i++)

{

curr=q.del(); //curr will store the value at the front of queue

for(int c=1;c<=vcnt;c++)

{

if(a[curr][c]==1) //check for the adjacent vertice of curr

{

q.insert(c); //insert adjacent vertices of curr in the queue

if(search(c)==0)

{

v[rcnt++]=c;

}

}

}

}

cout<<"BFS:";

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

{

cout<<v[i]<<" ";

}

cout<<endl;

}

void graph::dfs()

{

int sv,curr; //sv= start vertex

cout<<"Enter start vertex:";

cin>>sv;

v[rcnt++]=sv;

stack s;

s.push(sv);

for(int i=1;i<=vcnt;i++)

{

curr=s.pop(); //curr will store the value at the front of queue

for(int c=1;c<=vcnt;c++)

{

if(a[curr][c]==1) //check for the adjacent vertice of curr

{

s.push(c); //insert adjacent vertices of curr in the queue

if(search(c)==0)

{

v[rcnt++]=c;

}

}

}

}

cout<<"DFS:";

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

{

cout<<v[i]<<" ";

}

cout<<endl;

}

int main()

{

graph g;

int ch;

do{

cout<<"---MENU---\n";

cout<<"1.Insert Matrix\n";

cout<<"2.Display Matrix\n";

cout<<"3.BFS\n";

cout<<"4.DFS\n";

cout<<"5.Exit\n";

cout<<"Enter your choice\n";

cin>>ch;

switch(ch){

case 1:

{

g.admat();

break;

}

case 2:

{

g.dis\_admat();

break;

}

case 3:

{

g.bfs();

break;

}

case 4:

{

g.dfs();

break;

}

case 5:

{

cout<<"End of the program";

break;

}

default:

cout<<"Invalid choice!";

}

}while(ch!=5);

return 0;

}

output:

gescoe@gescoe-OptiPlex-3010:~/Desktop/SE-A-55$ g++ graph.cpp

gescoe@gescoe-OptiPlex-3010:~/Desktop/SE-A-55$ ./a.out

---MENU---

1. Insert Matrix
2. Display Matrix
3. BFS
4. DFS
5. Exit

Enter your choice

1

Enter total no. of vertices:4

Edge from 1 to 1:1

Edge from 1 to 2:2

Edge from 1 to 3:3

Edge from 1 to 4:4

Edge from 2 to 1:5

Edge from 2 to 2:6

Edge from 2 to 3:7

Edge from 2 to 4:8

Edge from 3 to 1:9

Edge from 3 to 2:10

Edge from 3 to 3:11

Edge from 3 to 4:12

Edge from 4 to 1:13

Edge from 4 to 2:14

Edge from 4 to 3:15

Edge from 4 to 4:16

---MENU---

1. Insert Matrix
2. Display Matrix
3. BFS
4. DFS
5. Exit

Enter your choice

2

1. 2 3 4

5 6 7 8

9 10 11 12

13 14 15 16

---MENU---

1. Insert Matrix
2. Display Matrix
3. BFS
4. DFS
5. Exit

Enter your choice

3

Enter start vertex:1

BFS:1

---MENU---

1. Insert Matrix
2. Display Matrix
3. BFS
4. DFS
5. Exit

Enter your choice

2

1. 2 3 4

5 6 7 8

9 10 11 12

13 14 15 16

---MENU---

1. Insert Matrix
2. Display Matrix
3. BFS
4. DFS
5. Exit

Enter your choice

4

Enter start vertex:2

Stack is empty.

Stack is empty.

Stack is empty.

DFS:1 2

---MENU---

1. Insert Matrix
2. Display Matrix
3. BFS
4. DFS
5. Exit

Enter your choice

2

1. 2 3 4

5 6 7 8

9 10 11 12

13 14 15 16

---MENU---

1. Insert Matrix
2. Display Matrix
3. BFS
4. DFS
5. Exit

Enter your choice

5

End of the [programgescoe@gescoe-OptiPlex-3010](mailto:programgescoe@gescoe-OptiPlex-3010):~/Desktop/SE-A-55$

Sure! Below is a detailed and comprehensive explanation of **Graph Theory**, **Breadth-First Search (BFS)**, and **Depth-First Search (DFS)**, including extended theoretical details, algorithmic steps, pseudocode, and applications, followed by an explanation of the code you provided.

### ****Graph Theory****

#### ****What is a Graph?****

A **graph** is a non-linear data structure consisting of nodes (also called vertices) and edges (connections between the nodes). A graph can be represented in various forms, such as an adjacency matrix, adjacency list, or edge list. Graphs are widely used to represent relationships in real-world problems such as social networks, maps, and organizational structures.

**Types of Graphs**:

* **Undirected Graph**: In an undirected graph, the edges do not have any direction. For example, a connection between two people in a social network is undirected.
* **Directed Graph (Digraph)**: In a directed graph, edges have a direction, i.e., each edge is directed from one vertex to another. A directed edge from vertex A to vertex B is represented as A → B.
* **Weighted Graph**: In a weighted graph, each edge has a weight or cost associated with it, representing the "distance" or "cost" between two vertices.
* **Unweighted Graph**: In an unweighted graph, all edges are considered to have the same weight or cost.
* **Cyclic Graph**: A cyclic graph contains a cycle, which is a path that starts and ends at the same vertex.
* **Acyclic Graph**: An acyclic graph does not contain any cycles. If a directed acyclic graph (DAG) has no directed cycles, it’s commonly used for scheduling or task precedence.

**Graph Representation**:

* **Adjacency Matrix**: A 2D array a[i][j] where each element is 1 if there is an edge between vertex i and vertex j, and 0 otherwise. For a graph with n vertices, the adjacency matrix will be an n × n matrix.
* **Adjacency List**: Each vertex has a list that contains all the vertices it is directly connected to.
* **Edge List**: A list of edges represented as pairs of vertices, i.e., [(u, v), (v, w)].

### ****Breadth-First Search (BFS)****

#### ****Theory of BFS****:

**Breadth-First Search (BFS)** is an algorithm for traversing or searching a graph in a level-by-level manner, i.e., it explores all neighbors at the present depth level before moving on to nodes at the next level.

BFS is commonly used to find the **shortest path** in an unweighted graph, traverse a tree or graph level-wise, and in various other applications.

##### ****Key Characteristics****:

* BFS uses a **queue** to manage the nodes to visit.
* It starts from a **source node** and explores all the neighboring nodes before moving to the next level of nodes.
* It guarantees that it finds the shortest path in an unweighted graph.

##### ****How BFS Works****:

1. **Initialize**:
   * Start from the source node, mark it as visited, and enqueue it into the queue.
2. **Processing**:
   * Dequeue a node from the front of the queue.
   * Visit all the unvisited adjacent nodes of the current node.
   * Mark each unvisited neighbor as visited and enqueue it.
3. **Termination**:
   * Continue the process until the queue is empty, meaning all reachable nodes have been visited.

##### ****Pseudocode for BFS****:

BFS(graph, start\_vertex):

Initialize a queue Q

Mark start\_vertex as visited

Enqueue start\_vertex into Q

While Q is not empty:

current\_vertex = Dequeue Q

Visit current\_vertex

For each neighbor of current\_vertex:

If neighbor is not visited:

Mark neighbor as visited

Enqueue neighbor into Q

##### ****Time Complexity****:

* **O(V + E)**: Where V is the number of vertices and E is the number of edges. Each vertex is processed once, and each edge is explored once.

##### ****Applications of BFS****:

1. **Shortest Path**: BFS can be used to find the shortest path in an unweighted graph.
2. **Level-order Traversal**: BFS is used in trees for level-order traversal.
3. **Web Crawlers**: BFS is useful for crawling websites level by level.
4. **Social Networks**: Finding the degree of separation between users.

### ****Depth-First Search (DFS)****

#### ****Theory of DFS****:

**Depth-First Search (DFS)** is an algorithm for traversing or searching a graph in a depthward motion. DFS explores as far as possible along a branch before backtracking.

DFS uses a **stack** (either explicitly or via recursion) to store the nodes to be visited.

##### ****Key Characteristics****:

* DFS explores nodes along a single path until it reaches a dead end or a leaf node.
* It then backtracks and explores the next unvisited adjacent node.
* DFS can be implemented using a **stack** or recursively.

##### ****How DFS Works****:

1. **Initialize**:
   * Start from the source node, mark it as visited, and push it onto the stack.
2. **Processing**:
   * Pop a node from the top of the stack.
   * Visit the node and explore all its unvisited neighbors.
   * Push the unvisited neighbors onto the stack.
3. **Termination**:
   * Continue this process until the stack is empty, meaning all reachable nodes have been visited.

##### ****Pseudocode for DFS****:

DFS(graph, start\_vertex):

Initialize a stack S

Mark start\_vertex as visited

Push start\_vertex onto S

While S is not empty:

current\_vertex = Pop S

Visit current\_vertex

For each neighbor of current\_vertex:

If neighbor is not visited:

Mark neighbor as visited

Push neighbor onto S

##### ****Time Complexity****:

* **O(V + E)**: Where V is the number of vertices and E is the number of edges. Each vertex and edge is explored once.

##### ****Applications of DFS****:

1. **Topological Sorting**: DFS is used in topological sorting of directed acyclic graphs (DAGs).
2. **Connected Components**: DFS helps in finding connected components in an undirected graph.
3. **Path Finding**: DFS can be used to find a path from the source to the destination in a graph.
4. **Cycle Detection**: DFS is useful for detecting cycles in a graph.

### ****Code Explanation (Graph Representation and Traversals)****

#### ****Graph Representation****:

In the provided code, the graph is represented using an **adjacency matrix**:

int a[20][20]; // Adjacency matrix for the graph

* The a[i][j] element is 1 if there is an edge from vertex i to vertex j, otherwise 0.

#### ****BFS****:

The **BFS** function is implemented as follows:

1. The user inputs the starting vertex.
2. A **queue** is used to store the vertices to visit.
3. The starting vertex is added to the queue, and the process continues until all reachable vertices are visited.

void graph::bfs() {

int sv, curr;

cout << "Enter start vertex:";

cin >> sv;

v[rcnt++] = sv; // Mark the start vertex as visited

queue q;

q.insert(sv); // Add start vertex to the queue

while (q.front != q.rear) {

curr = q.del(); // Dequeue the next vertex

for (int c = 1; c <= vcnt; c++) {

if (a[curr][c] == 1) { // If an edge exists from curr to c

q.insert(c); // Enqueue the adjacent vertex

if (search(c) == 0) {

v[rcnt++] = c; // Mark c as visited

}

}

}

}

cout << "BFS: ";

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

cout << v[i] << " "; // Print visited vertices

}

cout << endl;

}

#### ****DFS****:

The **DFS** function is implemented as follows:

1. The user inputs the starting vertex.
2. A **stack** is used to manage the traversal.
3. The process continues by visiting the adjacent vertices and pushing them to the stack.

void graph::dfs() {

int sv, curr;

cout << "Enter start vertex:";

cin >> sv;

v[rcnt++] = sv; // Mark the start vertex as visited

stack s;

s.push(sv); // Push the start vertex to the stack

while (s.top != -1) {

curr = s.pop(); // Pop the top vertex from the stack

for (int c = 1; c <= vcnt; c++) {

if (a[curr][c] == 1) { // If an edge exists from curr to c

s.push(c); // Push the adjacent vertex

if (search(c) == 0) {

v[rcnt++] = c; // Mark c as visited

}

}

}

}

cout << "DFS: ";

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

cout << v[i] << " "; // Print visited vertices

}

cout << endl;

}

### ****Conclusion****

* **BFS** and **DFS** are both fundamental graph traversal algorithms, each suited to different types of problems.
* **BFS** explores all neighbors level by level and guarantees the shortest path in an unweighted graph.
* **DFS** explores deeply into the graph and is useful for problems like cycle detection and topological sorting.
* The **adjacency matrix** representation is a simple and effective way to store graphs, especially when dealing with smaller graphs or dense graphs.

Let me know if you need any more details or clarifications!