Aim: Graph Traversal techniques.

Objective:
Relationship Representation: A graph models connections between entities (nodes) using edges, making it ideal for scenarios like social networks, web structures, and transportation systems.Algorithm Efficiency: Graphs support algorithms for pathfinding traversal (BFS, DFS), and network flow, enabling tasks like route optimization and resource management.Flexibility: Graphs allow dynamic modification with the ability to add or remove nodes and edges, making them adaptable to changing data structures.
Tools Used: Visual Studio Code.

Concept:
DFS explores a graph by starting at the root (or any arbitrary node) and proceeds as far down a branch as possible before backtracking. The algorithm uses a stack (either implicitly via recursion or explicitly) to remember the path it has taken. The key idea is to visit a node, then recursively visit its unvisited neighbors. This continues until there are no unvisited neighbors left, at which point the algorithm backtracks to the previous node and continues. DFS is typically used for solving problems that require exploring all possibilities, such as pathfinding, topological sorting, and connected components.
Depth-First Search (DFS) and Breadth-First Search (BFS) are two fundamental graph traversal algorithms used to explore all the nodes in a graph or tree. They differ in their approach to visiting nodes and are employed in various applications like finding paths, solving puzzles, and performing searches in artificial intelligence.

Algorithm:
DFS Algorithm (Using an Array for Visited Nodes)
1. Use an array visited[] of size V (number of vertices) where each element is initially set to False (indicating that the node has not been visited).
2. Start from a given node, mark it as visited by setting visited[node] = True.
3. Recursively visit each unvisited neighbor of the current node, marking each one as visited.
4. Once all neighbors of a node are visited, backtrack and continue the process until all nodes are visited.
BFS Algorithm using an Array
1. Use an array visited[] of size V (number of vertices) where each element is initially set to False (indicating that the node has not been visited).
2. Initialize a queue and enqueue the starting node.
3. While the queue is not empty:
    Dequeue a node and visit it (mark it as visited).
    Enqueue all the unvisited neighbors of the current node.
4. Repeat until all nodes reachable from the starting node have been visited.

Solution:
BFS
```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

class BFS
{
public:
   void bfsTraversal(int start, int nodes, const vector<vector<int>> &arr)
   {
```

```
        vector<bool> visited(nodes, false);
        queue<int> q;
        q.push(start);
        visited[start] = true;

        cout << "BFS Traversal: ";
        while (!q.empty())
        {
            int current = q.front();
            q.pop();
            cout << (char)('A' + current) << " ";

            for (int i = 0; i < nodes; i++)
            {
                if (arr[current][i] == 1 && !visited[i])
                {
                    q.push(i);
                    visited[i] = true;
                }
            }
        }
        cout << endl;
    }
};

int main()
{
    int nodes;
    cout << "How many nodes: ";
    cin >> nodes;

    vector<vector<int>> arr(nodes, vector<int>(nodes));
    for (int i = 0; i < nodes; i++)
    {
        for (int j = 0; j < nodes; j++)
        {
            cout << (char)('A' + i) << " to " << (char)('A' + j) << ": ";
            cin >> arr[i][j];
        }
    }

    char startNode;
    cout << "\nEnter the starting node : ";
    cin >> startNode;
    int startIndex = startNode - 'A';

    if (startIndex >= 0 && startIndex < nodes)
    {
        BFS bfs;
        bfs.bfsTraversal(startIndex, nodes, arr);
    }
    else
    {
        cout << "Invalid starting node." << endl;
    }
```

```
    return 0;
}
```

Output:

```
How many nodes: 6
A to A: 0
A to B: 1
A to C: 1
A to D: 1
A to E: 0
A to F: 0
B to A: 1
B to B: 0
B to C: 0
B to D: 0
B to E: 1
B to F: 0
C to A: 1
C to B: 0
C to C: 0
C to D: 0
C to E: 1
C to F: 0
D to A: 1
D to B: 0
D to C: 0
D to D: 0
D to E: 0
D to F: 1
E to A: 0
E to B: 1
E to C: 1
E to D: 0
E to E: 0
E to F: 1
F to A: 0
F to B: 0
F to C: 0
F to D: 1
F to E: 1
F to F: 0

Enter the starting node : A
BFS Traversal: A B C D E F
```

DFS

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

class DFS
{
```

```
public:
    void dfsTraversal(int start, int nodes, const vector<vector<int>> &arr)
    {
        vector<bool> visited(nodes, false);
        stack<int> s;
        s.push(start);

        cout << "DFS Traversal: ";
        while (!s.empty())
        {
            int current = s.top();
            s.pop();

            if (!visited[current])
            {
                cout << (char)('A' + current) << " ";
                visited[current] = true;
            }

            for (int i = nodes - 1; i >= 0; i--)
            {
                if (arr[current][i] == 1 && !visited[i])
                {
                    s.push(i);
                }
            }
        }
        cout << endl;
    }
};

int main()
{
    int nodes;
    cout << "How many nodes: ";
    cin >> nodes;

    vector<vector<int>> arr(nodes, vector<int>(nodes));
    for (int i = 0; i < nodes; i++)
    {
        for (int j = 0; j < nodes; j++)
        {
            cout << (char)('A' + i) << " to " << (char)('A' + j) << ": ";
            cin >> arr[i][j];
        }
    }

    char startNode;
    cout << "\nEnter the starting node: ";
    cin >> startNode;
    int startIndex = startNode - 'A';

    if (startIndex >= 0 && startIndex < nodes)
    {
        DFS dfs;
        dfs.dfsTraversal(startIndex, nodes, arr);
```

```
    }
    else
    {
        cout << "Invalid starting node." << endl;
    }

    return 0;
}
```

Output: How many nodes: 6
A to A: 0
A to B: 1
A to C: 1
A to D: 1
A to E: 0
A to F: 0
B to A: 1
B to B: 0
B to C: 0
B to D: 0
B to E: 1
B to F: 0
C to A: 1
C to B: 0
C to C: 0
C to D: 0
C to E: 1
C to F: 0
D to A: 1
D to B: 0
D to C: 0
D to D: 0
D to E: 0
D to F: 1
E to A: 0
E to B: 1
E to C: 1
E to D: 0
E to E: 0
E to F: 1
F to A: 0
F to B: 0
F to C: 0
F to D: 1
F to E: 1
F to F: 0

Enter the starting node: A
DFS Traversal: A B E C F D

Observation:

A graph is a data structure used to represent connections between entities, where entities are represented as vertices (nodes) and their relationships as edges (arcs). In the adjacency matrix representation, the graph is stored as a 2D array, with each cell indicating whether an edge exists between two vertices. This approach is efficient for dense graphs, as it allows quick edge lookups. However, it can be memory-intensive for sparse graphs with fewer edges. While adjacency matrices are simple and effective for certain operations, tasks like finding neighbors or traversing edges require scanning rows or columns, which can be less efficient for very large graphs.