Data Structures and Algorithms (Spring2024)

Week11/12 (16/16/22/23/23-May-2024)

M Ateeq,
Department of Data Science, The Islamia University of Bahawalpur.

Definition and Terminology: Vertices, Edges, Adjacency List, Adjacency Matrix

1. Vertices (Nodes)

- **Definition**: A vertex (plural: vertices) is a fundamental unit of a graph that represents an entity or a point. Vertices are often denoted as V in graph theory.
- **Example**: In a social network graph, each vertex can represent a person.

Notation:

$$V = \{v_1, v_2, v_3, \ldots, v_n\}$$

2. Edges (Links)

- **Definition**: An edge is a connection between two vertices in a graph. Edges can be directed (from one vertex to another) or undirected (bi-directional). They are often denoted as E.
- **Example**: In a social network graph, an edge can represent a friendship or a connection between two people.

Notation: $E = \{e_1, e_2, e_3, \ldots, e_m\}$

- **Directed Edge**: An edge with a direction from vertex u to vertex v is denoted as (u, v).
- **Undirected Edge**: An edge without direction between vertex u and vertex v is denoted as $\{u, v\}$.

3. Adjacency List

- **Definition**: An adjacency list represents a graph as an array of lists. The index of the array represents a vertex, and each element in the list represents the vertices connected to the vertex at that index.
- **Advantages**: Efficient in terms of space for sparse graphs.
- **Example**: For a graph with vertices A, B, C, D, where A is connected to B and C, and B is connected to D:

Graph Representation: $A \leftrightarrow B \leftrightarrow D\ A \leftrightarrow C$

Adjacency List:

```
A: B, C
```

- **Directed Graph Example**: For a directed graph with vertices A, B, C where $A \rightarrow B$ and $B \rightarrow C$:

Graph Representation: $A \rightarrow B \rightarrow C$

Adjacency List:

```
A: B
B: C
C:
```

4. Adjacency Matrix

- **Definition**: An adjacency matrix represents a graph as a 2D array (or matrix). The rows and columns represent vertices, and the cell at the intersection of row i and column j represents the edge between vertices i and j. If there is an edge, the cell contains 1 (or the weight of the edge); otherwise, it contains 0.
- **Advantages**: Efficient for dense graphs, quick to check if an edge exists.
- **Example**: For a graph with vertices A, B, C, D, where A is connected to B and C, and B is connected to D:

Graph Representation: $A \leftrightarrow B \leftrightarrow D\, A \leftrightarrow C$

Adjacency Matrix:

```
   A B C D
A [ 0 1 1 0 ]
B [ 1 0 0 1 ]
C [ 1 0 0 0 ]
D [ 0 1 0 0 ]
```

- **Directed Graph Example**: For a directed graph with vertices A, B, C where $A \rightarrow B$ and $B \rightarrow C$:

Graph Representation: $A \rightarrow B \rightarrow C$

Adjacency Matrix:

```
   A B C
A [ 0 1 0 ]
B [ 0 0 1 ]
C [ 0 0 0 ]
```

Summary of Key Terms

- **Vertex (Node)**: The fundamental unit of a graph.
- **Edge (Link)**: The connection between two vertices.
- **Adjacency List**: A space-efficient representation using lists.
- **Adjacency Matrix**: A matrix representation useful for dense graphs.

Types of Graphs: Directed, Undirected, Weighted, Unweighted

1. Directed Graphs (Digraphs)

- **Definition**: A directed graph is a set of vertices connected by edges, where the edges have a direction. This means each edge goes from one vertex to another specific vertex, indicated by an arrow.
- **Notation**: An edge from vertex u to vertex v is denoted as (u, v).
- **Example**: Consider a graph representing a set of web pages and links. If page A links to page B, but not vice versa, the graph would have a directed edge from A to B.

Graph Representation: $A \rightarrow B$ $B \rightarrow C$

- **Adjacency List**:

```
A: B
B: C
C:
```

- **Adjacency Matrix**:

```
  A B C
A [ 0 1 0 ]
B [ 0 0 1 ]
C [ 0 0 0 ]
```

2. Undirected Graphs

- **Definition**: An undirected graph is a set of vertices connected by edges, where the edges have no direction. Each edge simply connects two vertices, indicating a bi-directional relationship.
- **Notation**: An edge between vertex u and vertex v is denoted as $\{u, v\}$.
- **Example**: Consider a graph representing friendships in a social network. If person A is friends with person B, and person B is friends with person A, the graph would have an undirected edge between A and B.

Graph Representation: $A \leftrightarrow B$ $B \leftrightarrow C$

- **Adjacency List**:

```
A: B
B: A, C
C: B
```

- **Adjacency Matrix**:

```
  A B C
A [ 0 1 0 ]
B [ 1 0 1 ]
C [ 0 1 0 ]
```

3. Weighted Graphs

- **Definition**: A weighted graph is a graph in which each edge has a numerical value, called a weight, associated with it. This weight can represent various things like distance, cost, or time.
- **Notation**: An edge with weight w from vertex u to vertex v is denoted as (u, v, w).
- **Example**: Consider a graph representing a network of cities and roads between them, where the weight of each edge represents the distance between the cities.

Graph Representation: $A \overset{5}{\longrightarrow} B \; B \overset{2}{\longrightarrow} C$

- **Adjacency List**:

```
A: B (5)
B: C (2)
C:
```

- **Adjacency Matrix**:

```
  A B C
A [ 0 5 0 ]
B [ 0 0 2 ]
C [ 0 0 0 ]
```

4. Unweighted Graphs

- **Definition**: An unweighted graph is a graph in which all the edges have the same weight, typically considered to be 1. These graphs are often used when the presence of an edge is more important than its weight.
- **Example**: Consider a graph representing a simple network of computers where any direct connection between two computers is equally important.

Graph Representation: $A \leftrightarrow B \; B \leftrightarrow C$

- **Adjacency List**:

```
A: B
B: A, C
C: B
```

- **Adjacency Matrix**:

```
  A B C
A [ 0 1 0 ]
```

Summary of Key Types

- **Directed Graph**: Edges have direction (A → B).
- **Undirected Graph**: Edges are bi-directional (A ↔ B).
- **Weighted Graph**: Edges have weights (A → B with weight 5).
- **Unweighted Graph**: Edges are unweighted or have the same weight (A ↔ B).

Adjacency List and Matrix Representation in C++

```cpp
#include <iostream>
#include <vector>
#include <list>
using namespace std;

// Graph class using Adjacency List representation
class GraphList {
private:
    int numVertices;
    vector<list<int>> adjList;

public:
    GraphList(int vertices) {
        numVertices = vertices;
        adjList.resize(vertices);
    }



    void addEdge(int u, int v) {
        adjList[u].push_back(v);
        adjList[v].push_back(u); // For undirected graph
    }

    void printAdjList() {
        for (int i = 0; i < numVertices; ++i) {
            cout << i << ": ";
            for (int neighbor : adjList[i]) {
                cout << neighbor << " ";
            }
            cout << endl;
        }
    }
};
```

```
// Graph class using Adjacency Matrix representation
class GraphMatrix {
private:
    int numVertices;
    vector<vector<int>> adjMatrix;

public:
    GraphMatrix(int vertices) {
        numVertices = vertices;
        adjMatrix.resize(vertices, vector<int>(vertices, 0));
    }



    void addEdge(int u, int v) {
        adjMatrix[u][v] = 1;
        adjMatrix[v][u] = 1; // For undirected graph
    }

    void printAdjMatrix() {
        for (int i = 0; i < numVertices; ++i) {
            for (int j = 0; j < numVertices; ++j) {
                cout << adjMatrix[i][j] << " ";
            }
            cout << endl;
        }
    }
};



int main() {
    int vertices = 5;

    // Adjacency List representation
    GraphList graphList(vertices);
    graphList.addEdge(0, 1);
    graphList.addEdge(0, 2);
    graphList.addEdge(1, 3);
    graphList.addEdge(2, 4);
    cout << "Graph representation using Adjacency List:" << endl;
    graphList.printAdjList();

    cout << endl;
```

```
    // Adjacency Matrix representation
    GraphMatrix graphMatrix(vertices);
    graphMatrix.addEdge(0, 1);
    graphMatrix.addEdge(0, 2);
    graphMatrix.addEdge(1, 3);
    graphMatrix.addEdge(2, 4);
    cout << "Graph representation using Adjacency Matrix:" << endl;
    graphMatrix.printAdjMatrix();

    return 0;
}
```

Explanation

GraphList Class (Adjacency List Representation)

1. **Private Members**:
   - `int numVertices` : Stores the number of vertices in the graph.
   - `vector<list<int>> adjList` : A vector of lists where each list represents the adjacency list for a vertex.
2. **Public Methods**:
   - **Constructor**: Initializes `numVertices` and resizes `adjList` to have `vertices` number of elements.
   - **addEdge(int u, int v)**: Adds an undirected edge between vertices `u` and `v` by adding `v` to the adjacency list of `u` and `u` to the adjacency list of `v` .
   - **printAdjList()**: Prints the adjacency list representation of the graph.

GraphMatrix Class (Adjacency Matrix Representation)

1. **Private Members**:
   - `int numVertices` : Stores the number of vertices in the graph.
   - `vector<vector<int>> adjMatrix` : A 2D vector (matrix) where `adjMatrix[i][j]` is 1 if there is an edge between vertices `i` and `j` , and 0 otherwise.
2. **Public Methods**:
   - **Constructor**: Initializes `numVertices` and resizes `adjMatrix` to be a `vertices x vertices` matrix initialized with zeros.
   - **addEdge(int u, int v)**: Adds an undirected edge between vertices `u` and `v` by setting `adjMatrix[u][v]` and `adjMatrix[v][u]` to 1.
   - **printAdjMatrix()**: Prints the adjacency matrix representation of the graph.

Main Function

1. **Initialization**:

- Creates a graph with 5 vertices using both adjacency list and adjacency matrix representations.

2. **Adding Edges**:
   - Adds edges to both graph representations using the `addEdge` method.
3. **Printing Graphs**:
   - Prints the adjacency list and adjacency matrix representations of the graph using the `printAdjList` and `printAdjMatrix` methods, respectively.

User-Defined Graph Preferences in C++

```
#include <iostream>
#include <vector>
#include <list>
#include <limits>
using namespace std;

// Graph class using Adjacency List representation
class GraphList {
private:
    int numVertices;
    bool isDirected;
    bool isWeighted;
    vector<list<pair<int, int>>> adjList; // Pair of (neighbor, wei
ght)



public:
    GraphList(int vertices, bool directed, bool weighted) {
        numVertices = vertices;
        isDirected = directed;
        isWeighted = weighted;
        adjList.resize(vertices);
    }

    void addEdge(int u, int v, int weight = 1) {
        adjList[u].emplace_back(v, weight);
        if (!isDirected) {
            adjList[v].emplace_back(u, weight);
        }
    }
```

```
    void printAdjList() {
        for (int i = 0; i < numVertices; ++i) {
            cout << i << ": ";
            for (auto neighbor : adjList[i]) {
                cout << "(" << neighbor.first << ", " << neighbor.s
econd << ") ";
            }
            cout << endl;
        }
    }
};

// Graph class using Adjacency Matrix representation
class GraphMatrix {
private:
    int numVertices;
    bool isDirected;
    bool isWeighted;
    vector<vector<int>> adjMatrix;



public:
    GraphMatrix(int vertices, bool directed, bool weighted) {
        numVertices = vertices;
        isDirected = directed;
        isWeighted = weighted;
        adjMatrix.resize(vertices, vector<int>(vertices, numeric_li
mits<int>::max())); // Initialize with "infinity" for weighted grap
hs
    }

    void addEdge(int u, int v, int weight = 1) {
        adjMatrix[u][v] = weight;
        if (!isDirected) {
            adjMatrix[v][u] = weight;
        }
    }
```

```
    void printAdjMatrix() {
        for (int i = 0; i < numVertices; ++i) {
            for (int j = 0; j < numVertices; ++j) {
                if (adjMatrix[i][j] == numeric_limits<int>::max())
{
                    cout << "∞ ";
                } else {
                    cout << adjMatrix[i][j] << " ";
                }
            }
            cout << endl;
        }
    }
};



int main() {
    int vertices;
    char directedChar, weightedChar;
    bool directed, weighted;

    // Input for graph preferences
    cout << "Enter the number of vertices: ";
    cin >> vertices;
    cout << "Is the graph directed (y/n)? ";
    cin >> directedChar;
    directed = (directedChar == 'y' || directedChar == 'Y');
    cout << "Is the graph weighted (y/n)? ";
    cin >> weightedChar;
    weighted = (weightedChar == 'y' || weightedChar == 'Y');
```

```
    // Create graphs
    GraphList graphList(vertices, directed, weighted);
    GraphMatrix graphMatrix(vertices, directed, weighted);

    // Input for edges
    int u, v, weight;
    char addMore;
    do {
        cout << "Enter edge (u v";
        if (weighted) cout << " weight";
        cout << "): ";
        cin >> u >> v;
        if (weighted) {
            cin >> weight;
        } else {
            weight = 1; // Default weight for unweighted graphs
        }
        graphList.addEdge(u, v, weight);
        graphMatrix.addEdge(u, v, weight);



        cout << "Add more edges (y/n)? ";
        cin >> addMore;
    } while (addMore == 'y' || addMore == 'Y');

    // Print graph representations
    cout << "\nGraph representation using Adjacency List:" << endl;
    graphList.printAdjList();

    cout << "\nGraph representation using Adjacency Matrix:" << end
l;
    graphMatrix.printAdjMatrix();

    return 0;
}
```

Explanation

GraphList Class (Adjacency List Representation)

1. **Private Members**:
   - `int numVertices` : Stores the number of vertices in the graph.
   - `bool isDirected` : Indicates if the graph is directed.
   - `bool isWeighted` : Indicates if the graph is weighted.
   - `vector<list<pair<int, int>>> adjList` : A vector of lists where each list represents the adjacency list for a vertex. Each element in the list is a pair containing the neighbor and the weight of the edge.
2. **Public Methods**:

- **Constructor**: Initializes `numVertices`, `isDirected`, and `isWeighted`, and resizes `adjList` to have `vertices` number of elements.
- **addEdge(int u, int v, int weight = 1)**: Adds an edge between vertices `u` and `v` with the given weight. If the graph is undirected, it adds the reverse edge as well.
- **printAdjList()**: Prints the adjacency list representation of the graph.

GraphMatrix Class (Adjacency Matrix Representation)

1. **Private Members**:
   - `int numVertices`: Stores the number of vertices in the graph.
   - `bool isDirected`: Indicates if the graph is directed.
   - `bool isWeighted`: Indicates if the graph is weighted.
   - `vector<vector<int>> adjMatrix`: A 2D vector (matrix) where `adjMatrix[i][j]` is the weight of the edge between vertices `i` and `j`, initialized to "infinity" (max int) for weighted graphs.
2. **Public Methods**:
   - **Constructor**: Initializes `numVertices`, `isDirected`, and `isWeighted`, and resizes `adjMatrix` to be a `vertices x vertices` matrix initialized with "infinity" for weighted graphs.
   - **addEdge(int u, int v, int weight = 1)**: Adds an edge between vertices `u` and `v` with the given weight. If the graph is undirected, it adds the reverse edge as well.
   - **printAdjMatrix()**: Prints the adjacency matrix representation of the graph.

Main Function

1. **Input for Graph Preferences**:
   - Prompts the user to enter the number of vertices, and whether the graph is directed and/or weighted.
2. **Create Graphs**:
   - Creates instances of `GraphList` and `GraphMatrix` based on the user's preferences.
3. **Input for Edges**:
   - Prompts the user to enter edges (and weights if the graph is weighted). Continues to prompt until the user decides to stop.
4. **Print Graphs**:
   - Prints the adjacency list and adjacency matrix representations of the graph.

Basic Operations: Adding/Removing Vertices/Edges, Traversing Graphs

1. Adding Vertices

- **Definition**: Adding a vertex involves inserting a new node into the graph.

- **Example**: If we have an existing graph with vertices A, B, C and we want to add a new vertex D, the graph will be updated to include D.

- **Adjacency List Before Adding**:

```
A: B
B: A, C
C: B
```

- **Adjacency List After Adding**:

```
A: B
B: A, C
C: B
D:
```

2. Removing Vertices

- **Definition**: Removing a vertex involves deleting a node and all edges connected to it from the graph.
- **Example**: If we remove vertex B from the graph, all edges connected to B are also removed.

Graph Representation: $A \leftrightarrow B \leftrightarrow C \; D$

Graph Representation After Removal: $A \; C \; D$

- **Adjacency List Before Removing**:

```
A: B
B: A, C
C: B
D:
```

- **Adjacency List After Removing**:

```
A:
C:
D:
```

3. Adding Edges

- **Definition**: Adding an edge involves creating a connection between two vertices.
- **Example**: If we add an edge between A and D in the existing graph, the graph will reflect this new connection.

Graph Representation: $A \leftrightarrow B \leftrightarrow C \; D$

Graph Representation After Adding Edge: $A \leftrightarrow B \leftrightarrow C \; A \leftrightarrow D$

- **Adjacency List Before Adding**:

```
A: B
B: A, C
C: B
D:
```

- **Adjacency List After Adding**:

```
A: B, D
B: A, C
C: B
D: A
```

4. Removing Edges

- **Definition**: Removing an edge involves deleting the connection between two vertices.
- **Example**: If we remove the edge between A and B, the graph will no longer show this connection.

Graph Representation: $A \leftrightarrow B \leftrightarrow C\, A \leftrightarrow D$

Graph Representation After Removing Edge: $A \leftrightarrow D\, B \leftrightarrow C$

- **Adjacency List Before Removing**:

```
A: B, D
B: A, C
C: B
D: A
```

- **Adjacency List After Removing**:

```
A: D
B: C
C: B
D: A
```

Graph Traversal: Depth-First Search (DFS) and Breadth-First Search (BFS)

Graph traversal refers to the process of visiting all the nodes in a graph in a systematic manner. Two common methods for graph traversal are Depth-First Search (DFS) and Breadth-First Search (BFS).

Depth-First Search (DFS)

Explanation:

- DFS explores a graph by starting at a root node and exploring as far as possible along each branch before backtracking.
- It uses a stack data structure, either explicitly or via recursion, to keep track of the vertices to be visited.

Use Cases:

- Finding a path in a maze.
- Topological sorting.
- Solving puzzles with only one solution (e.g., Sudoku).

Algorithm:

1. Start at the root node.
2. Push the root node onto the stack.
3. While the stack is not empty:
    - Pop the top node from the stack.
    - If the node has not been visited:
        - Mark the node as visited.
        - Push all adjacent nodes that have not been visited onto the stack.

Code Example:

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

class Graph {
private:
    int numVertices;
    vector<vector<int>> adjList; // Adjacency list represented as a
vector of vectors



public:
    Graph(int vertices) {
        numVertices = vertices;
        adjList.resize(vertices);
    }

    void addEdge(int u, int v) {
        adjList[u].push_back(v);
        adjList[v].push_back(u); // For undirected graph
    }
```

```
    void DFS(int start) {
        vector<bool> visited(numVertices, false);
        stack<int> stack;
        stack.push(start);

        while (!stack.empty()) {
            int vertex = stack.top();
            stack.pop();

            if (!visited[vertex]) {
                cout << vertex << " ";
                visited[vertex] = true;
            }



            for (int neighbor : adjList[vertex]) {
                if (!visited[neighbor]) {
                    stack.push(neighbor);
                }
            }
        }
    }

    void printAdjList() {
        for (int i = 0; i < numVertices; ++i) {
            cout << i << ": ";
            for (int neighbor : adjList[i]) {
                cout << neighbor << " ";
            }
            cout << endl;
        }
    }
};
```

```
int main() {
    Graph graph(5); // Create a graph with 5 vertices (0 to 4)
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);

    cout << "Graph representation (Adjacency List):" << endl;
    graph.printAdjList();

    cout << "\nDFS starting from vertex 0: ";
    graph.DFS(0);

    return 0;
}
```

Line-by-Line Explanation:

1. **`#include <iostream>`** : Includes the input-output stream library necessary for printing output to the console.
2. **`#include <vector>`** : Includes the vector library for dynamic arrays.
3. **`#include <stack>`** : Includes the stack library for DFS.
4. **`using namespace std;`** : Declares the use of the standard namespace to avoid prefixing standard library names with `std::` .
5. **`class Graph {`** : Declares the `Graph` class.
6. **`private:`** : Specifies the access level for members of the `Graph` class to be private.
7. **`int numVertices;`** : Declares `numVertices` , an integer to store the number of vertices in the graph.
8. **`vector<vector<int>> adjList;`** : Declares `adjList` , a vector of vectors to represent the adjacency list.
9. **`public:`** : Specifies the access level for members of the `Graph` class to be public.
10. **`Graph(int vertices) {`** : Constructor for the `Graph` class that takes the number of vertices as a parameter.

11. **`numVertices = vertices;`** : Initializes `numVertices` with the given number of vertices.
12. **`adjList.resize(vertices);`** : Resizes the `adjList` to have `vertices` number of elements.
13. **`}`** : Closes the constructor.
14. **`void addEdge(int u, int v) {`** : Declares the `addEdge` method that takes two integers `u` and `v` as parameters, representing an edge between vertices `u` and `v` .
15. **`adjList[u].push_back(v);`** : Adds vertex `v` to the adjacency list of vertex `u` . This means there is an edge from `u` to `v` .
16. **`adjList[v].push_back(u);`** : Adds vertex `u` to the adjacency list of vertex `v` . This means there is an edge from `v` to `u` , making the graph undirected.
17. **`}`** : Closes the `addEdge` method.
18. **`void DFS(int start) {`** : Declares the `DFS` method for Depth-First Search that takes a starting vertex `start` as a parameter.

19. `vector<bool> visited(numVertices, false);` : Declares a vector `visited` to keep track of visited vertices, initialized to `false`.
20. `stack<int> stack;` : Declares a stack to manage the vertices to be visited.

21. `stack.push(start);` : Pushes the starting vertex onto the stack.
22. `while (!stack.empty()) {` : Enters a loop that continues until the stack is empty.
23. `int vertex = stack.top();` : Retrieves the top vertex from the stack.
24. `stack.pop();` : Removes the top vertex from the stack.
25. `if (!visited[vertex]) {` : Checks if the vertex has not been visited.
26. `cout << vertex << " ";` : Prints the vertex.
27. `visited[vertex] = true;` : Marks the vertex as visited.
28. `}` : Closes the if statement.
29. `for (int neighbor : adjList[vertex]) {` : Iterates over each neighboring vertex of the current vertex.
30. `if (!visited[neighbor]) {` : Checks if the neighboring vertex has not been visited.

31. `stack.push(neighbor);` : Pushes the neighboring vertex onto the stack.
32. `}` : Closes the if statement.
33. `}` : Closes the for loop.
34. `}` : Closes the while loop.
35. `}` : Closes the `DFS` method.
36. `void printAdjList() {` : Declares the `printAdjList` method, which prints the adjacency list representation of the graph.
37. `for (int i = 0; i < numVertices; ++i) {` : Iterates over each vertex in the graph.
38. `cout << i << ": ";` : Prints the current vertex followed by a colon and space.
39. `for (int neighbor : adjList[i]) {` : Iterates over each neighboring vertex in the adjacency list of the current vertex.
40. `cout << neighbor << " ";` : Prints each neighboring vertex followed by a space.

41. `}` : Closes the inner for loop.
42. `cout << endl;` : Prints a newline character to separate the adjacency list of different vertices.
43. `}` : Closes the outer for loop.
44. `}` : Closes the `printAdjList` method.
45. `};` : Closes the `Graph` class definition.
46. `int main() {` : Defines the `main` function, the entry point of the program.
47. `Graph graph(5);` : Creates an instance of the `Graph` class named `graph` with 5 vertices (0 to 4).
48. `graph.addEdge(0, 1);` : Adds an edge between vertices 0 and 1.
49. `graph.addEdge(0, 2);` : Adds an edge between vertices 0 and 2.
50. `graph.addEdge(1, 3);` : Adds an edge between vertices 1 and 3.
51. `graph.addEdge(2, 4);` : Adds an edge between vertices 2 and 4.
52. `cout << "Graph representation (Adjacency List):" << endl;` : Prints a header for the graph representation.
53. `graph.printAdjList();` : Calls the `printAdjList` method to print the adjacency list representation of the graph.
54. `cout << "\nDFS starting from vertex 0: ";` : Prints a header for DFS traversal.

55. `graph.DFS(0);` : Calls the `DFS` method to perform Depth-First Search starting from vertex 0.
56. `return 0;` : Returns 0 to indicate that the program ended successfully.
57. `}` : Closes the `main` function.

Breadth-First Search (BFS)

Explanation:

- BFS explores a graph by starting at a root node and exploring all its neighboring nodes at the present depth level before moving on to nodes at the next depth level.
- It uses a queue data structure to keep track of the vertices to be visited.

Use Cases:

- Finding the shortest path in an unweighted graph.
- Level-order traversal of a tree.
- Finding connected components in a graph.

Algorithm:

1. Start at the root node.
2. Enqueue the root node.
3. While the queue is not empty:
    - Dequeue the front node from the queue.
    - If the node has not been visited:
        - Mark the node as visited.
        - Enqueue all adjacent nodes that have not been visited.

Code Example:

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

class Graph {
private:
    int numVertices;
    vector<vector<int>> adjList; // Adjacency list represented as a
vector of vectors

public:
    Graph(int vertices) {
        numVertices = vertices;
        adjList.resize(vertices);
    }
```

```
    void addEdge(int u, int v) {
        adjList[u].push_back(v);
        adjList[v].push_back(u); // For undirected graph
    }

    void BFS(int start) {
        vector<bool> visited(numVertices, false);
        queue<int> queue;
        queue.push(start);
        visited[start] = true;



        while (!queue.empty()) {
            int vertex = queue.front();
            queue.pop();
            cout << vertex << " ";

            for (int neighbor : adjList[vertex]) {
                if (!visited[neighbor]) {
                    queue.push(neighbor);
                    visited[neighbor] = true;
                }
            }
        }
    }



    void printAdjList() {
        for (int i = 0; i < numVertices; ++i) {
            cout << i << ": ";
            for (int neighbor : adjList[i]) {
                cout << neighbor << " ";
            }
            cout << endl;
        }
    }
};
```

```
int main() {
    Graph graph(5); // Create a graph with 5 vertices (0 to 4)
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);

    cout << "Graph representation (Adjacency List):" << endl;
    graph.printAdjList();

    cout << "\nBFS starting from vertex 0: ";
    graph.BFS(0);

    return 0;
}
```

Line-by-Line Explanation:

1. **`#include <iostream>`** : Includes the input-output stream library necessary for printing output to the console.
2. **`#include <vector>`** : Includes the vector library for dynamic arrays.
3. **`#include <queue>`** : Includes the queue library for BFS.
4. **`using namespace std;`** : Declares the use of the standard namespace to avoid prefixing standard library names with `std::` .
5. **`class Graph {`** : Declares the `Graph` class.
6. **`private:`** : Specifies the access level for members of the `Graph` class to be private.
7. **`int numVertices;`** : Declares `numVertices` , an integer to store the number of vertices in the graph.
8. **`vector<vector<int>> adjList;`** : Declares `adjList` , a vector of vectors to represent the adjacency list.
9. **`public:`** : Specifies the access level for members of the `Graph` class to be public.
10. **`Graph(int vertices) {`** : Constructor for the `Graph` class that takes the number of vertices as a parameter.

11. **`numVertices = vertices;`** : Initializes `numVertices` with the given number of vertices.
12. **`adjList.resize(vertices);`** : Resizes the `adjList` to have `vertices` number of elements.
13. **`}`** : Closes the constructor.
14. **`void addEdge(int u, int v) {`** : Declares the `addEdge` method that takes two integers `u` and `v` as parameters, representing an edge between vertices `u` and `v` .
15. **`adjList[u].push_back(v);`** : Adds vertex `v` to the adjacency list of vertex `u` . This means there is an edge from `u` to `v` .
16. **`adjList[v].push_back(u);`** : Adds vertex `u` to the adjacency list of vertex `v` . This means there is an edge from `v` to `u` , making the graph undirected.
17. **`}`** : Closes the `addEdge` method.
18. **`void BFS(int start) {`** : Declares the `BFS` method for Breadth-First Search that takes a starting vertex `start` as a parameter.

19. `vector<bool> visited(numVertices, false);` : Declares a vector `visited` to keep track of visited vertices, initialized to `false`.
20. `queue<int> queue;` : Declares a queue to manage the vertices to be visited.

21. `queue.push(start);` : Pushes the starting vertex onto the queue.
22. `visited[start] = true;` : Marks the starting vertex as visited.
23. `while (!queue.empty()) {` : Enters a loop that continues until the queue is empty.
24. `int vertex = queue.front();` : Retrieves the front vertex from the queue.
25. `queue.pop();` : Removes the front vertex from the queue.
26. `cout << vertex << " ";` : Prints the vertex.
27. `for (int neighbor : adjList[vertex]) {` : Iterates over each neighboring vertex of the current vertex.
28. `if (!visited[neighbor]) {` : Checks if the neighboring vertex has not been visited.
29. `queue.push(neighbor);` : Pushes the neighboring vertex onto the queue.
30. `visited[neighbor] = true;` : Marks the neighboring vertex as visited.

31. `}` : Closes the if statement.
32. `}` : Closes the for loop.
33. `}` : Closes the while loop.
34. `}` : Closes the `BFS` method.
35. `void printAdjList() {` : Declares the `printAdjList` method, which prints the adjacency list representation of the graph.
36. `for (int i = 0; i < numVertices; ++i) {` : Iterates over each vertex in the graph.
37. `cout << i << ": ";` : Prints the current vertex followed by a colon and space.
38. `for (int neighbor : adjList[i]) {` : Iterates over each neighboring vertex in the adjacency list of the current vertex.
39. `cout << neighbor << " ";` : Prints each neighboring vertex followed by a space.
40. `}` : Closes the inner for loop.
41. `cout << endl;` : Prints a newline character to separate the adjacency list of different vertices.
42. `}` : Closes the outer for loop.
43. `}` : Closes the `printAdjList` method.
44. `};` : Closes the `Graph` class definition.
45. `int main() {` : Defines the `main` function, the entry point of the program.

46. `Graph graph(5);` : Creates an instance of the `Graph` class named `graph` with 5 vertices (0 to 4).
47. `graph.addEdge(0, 1);` : Adds an edge between vertices 0 and 1.
48. `graph.addEdge(0, 2);` : Adds an edge between vertices 0 and 2.
49. `graph.addEdge(1, 3);` : Adds an edge between vertices 1 and 3.
50. `graph.addEdge(2, 4);` : Adds an edge between vertices 2 and 4.
51. `cout << "Graph representation (Adjacency List):" << endl;` : Prints a header for the graph representation.
52. `graph.printAdjList();` : Calls the `printAdjList` method to print the adjacency list representation of the graph.
53. `cout << "\nBFS starting from vertex 0: ";` : Prints a header for BFS traversal.

54. `graph.BFS(0);` : Calls the `BFS` method to perform Breadth-First Search starting from vertex 0.
55. `return 0;` : Returns 0 to indicate that the program ended successfully.

Summary

DFS:

- Uses a stack (either explicitly or via recursion).
- Explores as far as possible along each branch before backtracking.
- Useful for tasks like solving puzzles and topological sorting.

BFS:

- Uses a queue.
- Explores all neighbors at the present depth level before moving on to the next level.
- Useful for finding the shortest path in an unweighted graph and level-order traversal of trees.

Shortest Path Problem

The shortest path problem involves finding the shortest path from a starting node (source) to one or more target nodes (destinations) in a graph. This graph can be either directed or undirected, and it may have weights on its edges, which represent the cost of traveling between nodes.

Applications

1. **Navigation Systems:** GPS and other navigation systems use shortest path algorithms to provide the quickest or least congested route to a destination.
2. **Network Routing Protocols:** In computer networks, routing protocols like OSPF and BGP utilize shortest path calculations to optimize data packet delivery across the network.
3. **Social Networking:** Finding the shortest path in social networking graphs can help in identifying the minimum number of connections between individuals.
4. **Urban Planning:** City planners use shortest path calculations for efficient public transport systems and road networks.
5. **Supply Chain Logistics:** Optimizing the routes for delivery trucks to minimize travel time and fuel consumption.

Algorithm and Implementation

For the implementation, we'll use Dijkstra's algorithm, which is efficient for graphs with non-negative edge weights. Here's a breakdown of the algorithm:

1. Assign a tentative distance value to every node: zero for the initial node and infinity for all other nodes.
2. Set the initial node as current and mark it visited.
3. For the current node, consider all of its unvisited neighbors. Calculate their tentative distances through the current node. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.

4. When we've considered all of the unvisited neighbors of the current node, mark the current node as done. A node is marked as done if the smallest tentative distance among its unvisited neighbors has been found.
5. Select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node," and repeat from step 3.
6. Continue this process until all nodes are marked done.

C++ Implementation Using Adjacency List

We'll use a vector of pairs to represent the adjacency list, where each pair consists of an integer node and a weight. This implementation avoids using `unordered_map`, sticking to vectors for simplicity.

```
#include <iostream>
#include <vector>
#include <utility>
#include <queue>
#include <limits>

using namespace std;



// Define the adjacency list using a vector of vector of pairs
typedef pair<int, int> pii; // First element is the destination nod
e, second is the weight
vector<vector<pii>> adjList;

vector<int> dijkstra(int start, int n) {
    priority_queue<pii, vector<pii>, greater<pii>> pq; // Min-heap
to get the node with the smallest distance
    vector<int> dist(n, numeric_limits<int>::max()); // Distance va
lues

    // Start from the source node
    pq.push(make_pair(0, start));
    dist[start] = 0;

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();
```

```
            // Visit each adjacent vertex v
            for (auto & p : adjList[u]) {
                int v = p.first;
                int weight = p.second;

                // Calculate the new distance
                if (dist[v] > dist[u] + weight) {
                    dist[v] = dist[u] + weight;
                    pq.push(make_pair(dist[v], v));
                }
            }
        }

        return dist;
    }



    int main() {
        int n, m; // n is the number of vertices, m is the number of ed
    ges
        cin >> n >> m;
        adjList.resize(n);

        for (int i = 0; i < m; i++) {
            int u, v, w;
            cin >> u >> v >> w; // Read edge and weight
            adjList[u].push_back(make_pair(v, w));
            // For undirected graph, add the reverse edge
            // adjList[v].push_back(make_pair(u, w)); // Uncomment this
    line for undirected graph
        }



        int start = 0; // Assuming starting node as node 0
        vector<int> distances = dijkstra(start, n);

        // Printing the shortest distances from the start
        cout << "Shortest distances from node " << start << ":" << end
    l;
        for (int i = 0; i < n; i++) {
            cout << "Node " << i << ": " << distances[i] << endl;
        }

        return 0;
    }
```

This C++ program implements Dijkstra's algorithm using a priority queue to efficiently find the shortest path from a given start node to all other nodes in a weighted graph.

Make sure to provide input in the form of number of nodes and edges followed by each edge with its weight. For an undirected graph, you can uncomment the specified line to add

Here's a line-by-line explanation of the C++ code provided for Dijkstra's algorithm using an adjacency list:

1. **Header Includes:** We include the necessary C++ libraries:
   - `iostream` for input/output.
   - `vector` for using vectors.
   - `utility` for pairs.
   - `queue` for priority queues.
   - `limits` for numeric limits.
2. **Namespace Usage:** `using namespace std;` allows us to use standard library components without specifying `std::` .
3. **Type Definitions:**
   - `typedef pair<int, int> pii;` creates a shorthand for `pair<int, int>` called `pii` , where the first `int` is a destination node, and the second is the weight of the edge.
4. **Global Variables:**
   - `vector<vector<pii>> adjList;` defines a global variable `adjList` which is a vector of vectors of pairs. Each element of the main vector represents a node and contains a vector of pairs, where each pair is a neighboring node and the weight of the edge connecting them.
5. **Function `dijkstra` :** This function implements Dijkstra's algorithm:
   - Parameters are `start` (starting node) and `n` (total number of nodes).
   - `priority_queue<pii, vector<pii>, greater<pii>> pq;` declares a min-heap priority queue where the smallest element comes to the top.
   - `vector<int> dist(n, numeric_limits<int>::max());` initializes the distance to each node as infinity (`max` value of `int`).
   - `pq.push(make_pair(0, start));` and `dist[start] = 0;` sets the starting node's distance to 0 and pushes it onto the queue.
6. **Main Loop of Dijkstra's Algorithm:**
   - While the priority queue is not empty:
     - `int u = pq.top().second;` gets the node `u` with the smallest distance.
     - `pq.pop();` removes this node from the priority queue.
     - For each neighbor `v` of `u` with weight `weight` :
       - If the distance to `v` through `u` is smaller than the current distance to `v` , update `dist[v]` and push `v` onto the priority queue.
7. **Main Function:**
   - `int n, m;` declares `n` (number of vertices) and `m` (number of edges).
   - Input is taken for `n` and `m` .
   - `adjList.resize(n);` resizes `adjList` to hold `n` nodes.

- The edges are input in a loop, updating `adjList` accordingly. If the graph were undirected, a reverse edge would also be added.
- The `dijkstra` function is called from the `main` function, starting from node 0.

Reflection MCQs

Depth-First Search (DFS)

1. **Which data structure is used to implement Depth-First Search (DFS)?**
    - A. Queue
    - B. Stack
    - C. Priority Queue
    - D. Linked List
    - **Correct Answer: B. Stack**
2. **In DFS, how is the graph traversed?**
    - A. Level by level
    - B. As far as possible along each branch before backtracking
    - C. By visiting all nodes at the present depth level before moving on to the nodes at the next depth level
    - D. Randomly
    - **Correct Answer: B. As far as possible along each branch before backtracking**
3. **Which of the following is a use case of DFS?**
    - A. Finding the shortest path in an unweighted graph
    - B. Web crawling
    - C. Solving puzzles like mazes
    - D. Task scheduling
    - **Correct Answer: C. Solving puzzles like mazes**

Breadth-First Search (BFS)

4. **Which data structure is used to implement Breadth-First Search (BFS)?**
    - A. Stack
    - B. Queue
    - C. Priority Queue
    - D. Linked List
    - **Correct Answer: B. Queue**
5. **In BFS, how is the graph traversed?**
    - A. As far as possible along each branch before backtracking
    - B. By visiting all nodes at the present depth level before moving on to the nodes at the next depth level
    - C. Randomly
    - D. By following the shortest path
    - **Correct Answer: B. By visiting all nodes at the present depth level before moving on to the nodes at the next depth level**
6. **Which of the following is a use case of BFS?**
    - A. Finding the shortest path in an unweighted graph

- B. Topological sorting
- C. Solving puzzles like Sudoku
- D. Detecting cycles in a graph
- **Correct Answer: A. Finding the shortest path in an unweighted graph**

Comparison of DFS and BFS

7. **Which of the following statements is true about DFS and BFS?**
   - A. DFS is more suitable for finding the shortest path in an unweighted graph.
   - B. BFS uses a stack, while DFS uses a queue.
   - C. DFS is implemented using recursion or an explicit stack, while BFS uses a queue.
   - D. BFS is more suitable for detecting cycles in a graph.
   - **Correct Answer: C. DFS is implemented using recursion or an explicit stack, while BFS uses a queue**
8. **Which algorithm would you use to find the shortest path in an unweighted graph?**
   - A. DFS
   - B. BFS
   - C. Dijkstra's Algorithm
   - D. Kruskal's Algorithm
   - **Correct Answer: B. BFS**
9. **Which algorithm explores as far as possible along each branch before backtracking?**
   - A. BFS
   - B. Dijkstra's Algorithm
   - C. Prim's Algorithm
   - D. DFS
   - **Correct Answer: D. DFS**

Dijkstra's Algorithm

10. **What is the primary use of Dijkstra's algorithm?**
    - A. To find the shortest path in an unweighted graph
    - B. To find the shortest path in a weighted graph with non-negative weights
    - C. To detect cycles in a graph
    - D. To perform topological sorting
    - **Correct Answer: B. To find the shortest path in a weighted graph with non-negative weights**
11. **What data structure is commonly used in Dijkstra's algorithm to select the next vertex with the smallest known distance?**
    - A. Stack
    - B. Queue
    - C. Priority Queue (Min-Heap)
    - D. Linked List
    - **Correct Answer: C. Priority Queue (Min-Heap)**
12. **Which of the following statements is true about Dijkstra's algorithm?**

- A. It works with negative weights.
- B. It cannot handle graphs with cycles.
- C. It always finds the shortest path in a graph with non-negative weights.
- D. It uses a stack to keep track of the vertices.
- **Correct Answer: C. It always finds the shortest path in a graph with non-negative weights**

Depth-First Search (DFS) Examples

13. **Which graph traversal method is best for solving a maze where you need to find a path to the exit?**
    - A. BFS
    - B. DFS
    - C. Dijkstra's Algorithm
    - D. Kruskal's Algorithm
    - **Correct Answer: B. DFS**
14. **In a DFS implementation, what condition indicates you should backtrack?**
    - A. When all neighboring nodes are visited
    - B. When the shortest path is found
    - C. When a cycle is detected
    - D. When all nodes at the current level are visited
    - **Correct Answer: A. When all neighboring nodes are visited**
15. **What is the space complexity of DFS for a graph with (V) vertices and (E) edges?**
    - A. O(V)
    - B. O(E)
    - C. O(V + E)
    - D. O(V^2)
    - **Correct Answer: C. O(V + E)**

Breadth-First Search (BFS) Examples

16. **Which traversal method is best for finding the shortest path in an unweighted grid?**
    - A. BFS
    - B. DFS
    - C. Dijkstra's Algorithm
    - D. Prim's Algorithm
    - **Correct Answer: A. BFS**
17. **In a BFS implementation, what is the purpose of the queue?**
    - A. To store the nodes that need to be visited
    - B. To keep track of visited nodes
    - C. To store the shortest path
    - D. To maintain the depth of nodes
    - **Correct Answer: A. To store the nodes that need to be visited**
18. **What is the time complexity of BFS for a graph with (V) vertices and (E) edges?**
    - A. O(V)

- B. O(E)
- C. O(V + E)
- D. O(V^2)
- **Correct Answer: C. O(V + E)**

Dijkstra's Algorithm Examples

19. **Which data structure is essential for Dijkstra’s algorithm to work efficiently?**

    - A. Stack
    - B. Queue
    - C. Priority Queue
    - D. Linked List
    - **Correct Answer: C. Priority Queue**
20. **What does Dijkstra’s algorithm primarily rely on to find the shortest path?**

    - A. Negative edge weights
    - B. Breadth-first search
    - C. The smallest known distance to a vertex
    - D. Depth-first search
    - **Correct Answer: C. The smallest known distance to a vertex**
21. **Why is Dijkstra’s algorithm not suitable for graphs with negative weight edges?**

    - A. It can result in infinite loops
    - B. It assumes all edges have the same weight
    - C. It will not find the shortest path correctly
    - D. It only works on unweighted graphs
    - **Correct Answer: C. It will not find the shortest path correctly**

Make sure that you have:

- **completed all the concepts**
- **ran all code examples yourself**
- **tried changing little things in the code to see the impact**
- **implemented the required programs**

Please feel free to share your problems to ensure that your weekly tasks are completed and you are not staying behind