Graph Traversal in Grids (BFS & DFS)

Topics Covered

Topic	Approach	Traversal	Language	Notes
Graph Representations	Various	-	C++ / Java	Adjacency Matrix, List, Map
Number of Islands	BFS / DFS	Matrix	C++ / Java	Count components of '1'
Flood Fill	BFS	Matrix	C++	Classic 4-dir traversal
Rotten Oranges	BFS with time	Matrix	C++	BFS with time as 3rd param
Find Circle Number	DFS	Adjacency Matrix	C++	Convert to graph + DFS

Graph Representations

Overview

Different ways to represent graphs in memory for efficient traversal and operations.

Remember:

- **Adjacency Matrix**: Good for dense graphs, O(1) edge lookup
- **Adjacency List**: Good for sparse graphs, space efficient
- **Adjacency Map**: Flexible, handles non-contiguous node IDs

---

Adjacency Matrix - C++

```cpp
class GraphMatrix {
private:
    vector<vector<int>> adj;
    int vertices;

public:
    GraphMatrix(int v) : vertices(v) {
        adj.resize(v, vector<int>(v, 0));
    }

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

    void removeEdge(int u, int v) {
        adj[u][v] = 0;
        adj[v][u] = 0;
    }

    bool hasEdge(int u, int v) {
        return adj[u][v] == 1;
    }

    void printGraph() {
        for(int i = 0; i < vertices; i++) {
            for(int j = 0; j < vertices; j++) {
                cout << adj[i][j] << " ";
            }
            cout << "\n";
        }
    }
};
```

Adjacency Matrix - Java

```java
class GraphMatrix {
    private int[][] adj;
    private int vertices;

    public GraphMatrix(int v) {
        this.vertices = v;
        this.adj = new int[v][v];
```

```
    }

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

    public void removeEdge(int u, int v) {
        adj[u][v] = 0;
        adj[v][u] = 0;
    }

    public boolean hasEdge(int u, int v) {
        return adj[u][v] == 1;
    }

    public void printGraph() {
        for(int i = 0; i < vertices; i++) {
            for(int j = 0; j < vertices; j++) {
                System.out.print(adj[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Adjacency List - C++

```
class GraphList {
private:
    vector<vector<int>> adj;
    int vertices;

public:
    GraphList(int v) : vertices(v) {
        adj.resize(v);
    }

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

    void removeEdge(int u, int v) {
```

```
        adj[u].erase(remove(adj[u].begin(), adj[u].end(), v),
        adj[u].end());
        adj[v].erase(remove(adj[v].begin(), adj[v].end(), u),
        adj[v].end());
    }

    vector<int> getNeighbors(int u) {
        return adj[u];
    }

    void printGraph() {
        for(int i = 0; i < vertices; i++) {
            cout << i << ": ";
            for(int neighbor : adj[i]) {
                cout << neighbor << " ";
            }
            cout << "\n";
        }
    }
};
```

Adjacency List - Java

```
class GraphList {
    private ArrayList<ArrayList<Integer>> adj;
    private int vertices;

    public GraphList(int v) {
        this.vertices = v;
        this.adj = new ArrayList<>();
        for(int i = 0; i < v; i++) {
            adj.add(new ArrayList<>());
        }
    }

    public void addEdge(int u, int v) {
        adj.get(u).add(v);
        adj.get(v).add(u); // For undirected graph
    }

    public void removeEdge(int u, int v) {
        adj.get(u).remove(Integer.valueOf(v));
        adj.get(v).remove(Integer.valueOf(u));
    }
```

```
    public ArrayList<Integer> getNeighbors(int u) {
        return adj.get(u);
    }

    public void printGraph() {
        for(int i = 0; i < vertices; i++) {
            System.out.print(i + ": ");
            for(int neighbor : adj.get(i)) {
                System.out.print(neighbor + " ");
            }
            System.out.println();
        }
    }
}
```

Adjacency Map - C++

```
class GraphMap {
private:
    unordered_map<int, vector<int>> adj;

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

    void removeEdge(int u, int v) {
        if(adj.find(u) != adj.end()) {
            adj[u].erase(remove(adj[u].begin(), adj[u].end(), v),
        adj[u].end());
        }
        if(adj.find(v) != adj.end()) {
            adj[v].erase(remove(adj[v].begin(), adj[v].end(), u),
        adj[v].end());
        }
    }

    vector<int> getNeighbors(int u) {
        return adj[u];
    }

    bool hasNode(int u) {
        return adj.find(u) != adj.end();
    }
```

```
    void printGraph() {
        for(auto& [node, neighbors] : adj) {
            cout << node << ": ";
            for(int neighbor : neighbors) {
                cout << neighbor << " ";
            }
            cout << "\n";
        }
    }
};
```

Adjacency Map - Java

```
class GraphMap {
    private HashMap<Integer, ArrayList<Integer>> adj;

    public GraphMap() {
        this.adj = new HashMap<>();
    }

    public void addEdge(int u, int v) {
        adj.computeIfAbsent(u, k -> new ArrayList<>()).add(v);
        adj.computeIfAbsent(v, k -> new
        ArrayList<>()).add(u); // For undirected graph
    }

    public void removeEdge(int u, int v) {
        if(adj.containsKey(u)) {
            adj.get(u).remove(Integer.valueOf(v));
        }
        if(adj.containsKey(v)) {
            adj.get(v).remove(Integer.valueOf(u));
        }
    }

    public ArrayList<Integer> getNeighbors(int u) {
        return adj.getOrDefault(u, new ArrayList<>());
    }

    public boolean hasNode(int u) {
        return adj.containsKey(u);
    }

    public void printGraph() {
```

```
        for(Map.Entry<Integer, ArrayList<Integer>> entry :
        adj.entrySet()) {
            System.out.print(entry.getKey() + ": ");
            for(int neighbor : entry.getValue()) {
                System.out.print(neighbor + " ");
            }
            System.out.println();
        }
    }
}
```

Number of Islands

Approach

- Use BFS or DFS to traverse all '1's and mark them visited.
- Increment counter for each new BFS/DFS initiation.

Remember:

- Traverse in 4 directions.
- Visited array or mark visited in-place.

DFS - C++

```cpp
void dfs(int i, int j, vector<vector<char>>& grid,
         vector<vector<bool>>& visited, int m, int n) {
    if(i < 0 || i >= m || j < 0 || j >= n || visited[i][j] ||
         grid[i][j] == '0') return;

    visited[i][j] = true;
    dfs(i+1, j, grid, visited, m, n);
    dfs(i-1, j, grid, visited, m, n);
    dfs(i, j+1, grid, visited, m, n);
    dfs(i, j-1, grid, visited, m, n);
}

int numIslands(vector<vector<char>>& grid) {
    int m = grid.size(), n = grid[0].size();
    vector<vector<bool>> visited(m, vector<bool>(n, false));
    int count = 0;

    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) {
```

```
            if(grid[i][j] == '1' && !visited[i][j]) {
                dfs(i, j, grid, visited, m, n);
                count++;
            }
        }
    }
    return count;
}
```

DFS - Java

```
void dfs(int i, int j, char[][] grid, boolean[][] visited) {
    int m = grid.length, n = grid[0].length;
    if(i < 0 || i >= m || j < 0 || j >= n || visited[i][j] ||
        grid[i][j] == '0') return;

    visited[i][j] = true;
    dfs(i+1, j, grid, visited);
    dfs(i-1, j, grid, visited);
    dfs(i, j+1, grid, visited);
    dfs(i, j-1, grid, visited);
}

public int numIslands(char[][] grid) {
    int m = grid.length, n = grid[0].length;
    boolean[][] visited = new boolean[m][n];
    int count = 0;

    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) {
            if(grid[i][j] == '1' && !visited[i][j]) {
                dfs(i, j, grid, visited);
                count++;
            }
        }
    }
    return count;
}
```

BFS - C++

```
void bfs(int row, int col, vector<vector<char>>& grid,
         vector<vector<bool>>& vis, int m, int n) {
    queue<pair<int,int>> q;
    q.push({row, col});
```

```
    vis[row][col] = true;

    int dir[] = {1, 0, -1, 0, 1};

    while(!q.empty()) {
        auto [x, y] = q.front();
        q.pop();

        for(int i = 0; i < 4; i++) {
            int nx = x + dir[i];
            int ny = y + dir[i+1];

            if(nx >= 0 && nx < m && ny >= 0 && ny < n &&
               !vis[nx][ny] && grid[nx][ny] == '1') {
                vis[nx][ny] = true;
                q.push({nx, ny});
            }
        }
    }
}

int numIslands(vector<vector<char>>& grid) {
    int m = grid.size(), n = grid[0].size();
    vector<vector<bool>> visited(m, vector<bool>(n, false));
    int count = 0;

    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) {
            if(grid[i][j] == '1' && !visited[i][j]) {
                bfs(i, j, grid, visited, m, n);
                count++;
            }
        }
    }
    return count;
}
```

Flood Fill Algorithm

Approach

- BFS starting from (sr, sc)
- Recolor connected pixels with same original color.

C++ Code

```cpp
void bfs(int row, int col, vector<vector<int>>& image,
         vector<vector<bool>>& visited, int m, int n, int color)
        {
    queue<pair<int,int>> q;
    int orig = image[row][col];
    image[row][col] = color;
    visited[row][col] = true;
    q.push({row, col});

    int dir[] = {1, 0, -1, 0, 1};

    while(!q.empty()) {
        auto [r, c] = q.front();
        q.pop();

        for(int i = 0; i < 4; i++) {
            int nr = r + dir[i], nc = c + dir[i+1];
            if(nr >= 0 && nr < m && nc >= 0 && nc < n &&
               !visited[nr][nc] && image[nr][nc] == orig) {
                visited[nr][nc] = true;
                image[nr][nc] = color;
                q.push({nr, nc});
            }
        }
    }
}

vector<vector<int>> floodFill(vector<vector<int>>& image, int sr,
        int sc, int color) {
    int m = image.size(), n = image[0].size();
    if(image[sr][sc] == color) return image;

    vector<vector<bool>> visited(m, vector<bool>(n, false));
    bfs(sr, sc, image, visited, m, n, color);
    return image;
}
```

Rotten Oranges (With Time Tracking)

Approach

- BFS with 3rd param as time.

- Track how long it takes for all fresh oranges to rot.

C++ Code

```cpp
int orangesRotting(vector<vector<int>>& grid) {
    int m = grid.size(), n = grid[0].size();
    queue<pair<pair<int,int>, int>> q;
    vector<vector<bool>> visited(m, vector<bool>(n, false));
    int fresh = 0;

    // Initialize queue with all rotten oranges
    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) {
            if(grid[i][j] == 2) {
                q.push({{i,j}, 0});
                visited[i][j] = true;
            } else if(grid[i][j] == 1) {
                fresh++;
            }
        }
    }

    int time = 0;
    int dir[] = {1, 0, -1, 0, 1};

    while(!q.empty()) {
        auto [pos, t] = q.front();
        q.pop();
        auto [x, y] = pos;
        time = max(time, t);

        for(int i = 0; i < 4; i++) {
            int nx = x + dir[i], ny = y + dir[i+1];
            if(nx >= 0 && nx < m && ny >= 0 && ny < n &&
               !visited[nx][ny] && grid[nx][ny] == 1) {
                visited[nx][ny] = true;
                grid[nx][ny] = 2;
                q.push({{nx, ny}, t + 1});
                fresh--;
            }
        }
    }

    return fresh == 0 ? time : -1;
}
```

Find Circle Number (Connected Components in Graph)

Approach

- Convert adjacency matrix to graph
- Count connected components using DFS

C++ Code

```cpp
void dfs(int node, map<int, vector<int>>& adj, vector<bool>&
        visited) {
    visited[node] = true;
    for(int neighbor : adj[node]) {
        if(!visited[neighbor]) {
            dfs(neighbor, adj, visited);
        }
    }
}

int findCircleNum(vector<vector<int>>& isConnected) {
    int n = isConnected.size();
    map<int, vector<int>> adj;

    // Build adjacency list from matrix
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            if(isConnected[i][j] == 1 && i != j) {
                adj[i].push_back(j);
            }
        }
    }

    vector<bool> visited(n, false);
    int count = 0;

    for(int i = 0; i < n; i++) {
        if(!visited[i]) {
            dfs(i, adj, visited);
            count++;
        }
    }
```

```
    return count;
}
```

⚠ Common Mistakes to Avoid

- **Incorrect direction indices**: `dir[] = {1, 0, -1, 0, 1}` (make sure `dir[i]`, `dir[i+1]`)
- **Wrong bounds**: Check `i < m` and `j < n` correctly
- **DFS base case ordering**: `if(i < 0 || ...` must be before array access
- **Time tracking**: Not tracking time properly in BFS with levels
- **Visited check**: Forgetting to mark visited before adding to queue in BFS

Tips for Grid Problems

1. **Use visited matrix** or mark grid itself to avoid revisiting
2. **4-direction vector**: `dir[] = {1,0,-1,0,1}` for easy iteration
3. **Time tracking in BFS**: Use `queue<pair<pair<int,int>, int>>` or similar structure
4. **Learn both DFS and BFS** — useful in different scenarios
5. **Boundary checks first** — always validate indices before array access
6. **Multi-source BFS** — Add all initial sources to queue for problems like rotten oranges

---

Space & Time Complexity

Algorithm	Time Complexity	Space Complexity	Use Case
DFS	O(V + E)	O(V)	Path finding, cycle detection
BFS	O(V + E)	O(V)	Shortest path, level-order
Adjacency Matrix	O(1) lookup	$O(V^2)$	Dense graphs
Adjacency List	O(degree) lookup	O(V + E)	Sparse graphs

Algorithm	Time Complexity	Space Complexity	Use Case
Adjacency Map	O(1) avg lookup	O(V + E)	Dynamic graphs