Graph & Grid Problems Summary - July 28, 2025

1. 01 Matrix (LeetCode 542)

Problem: Given a binary matrix, return the distance of each cell to the nearest 0.

Approach: Multi-source BFS - Start BFS from all 0s simultaneously, setting their distance as 0 - Push all 0s into the queue initially - Explore 4 directions from each cell, setting neighbor's distance = current + 1

```
class Solution {
public:
    vector<vector<int>> updateMatrix(vector<vector<int>>& mat) {
        int m = mat.size(), n = mat[0].size();
        vector<vector<bool>> visited(m, vector<bool>(n, false));
        vector<vector<int>> dist(m, vector<int>(n));
        queue<tuple<int, int, int>> q;

        // Add all 0s to queue
        for(int i = 0; i < m; i++) {
            for(int j = 0; j < n; j++) {
                if(mat[i][j] == 0) {
                    q.push({i, j, 0});
                    visited[i][j] = true;
                }
            }
        }

        int dir[] = {1, 0, -1, 0, 1};
        while(!q.empty()) {
            auto [r, c, d] = q.front();
            q.pop();
            dist[r][c] = d;

            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] && mat[nr][nc] == 1) {
                    visited[nr][nc] = true;
                    q.push({nr, nc, d + 1});
                }
            }
        }
        return dist;
    }
};
```

2. Surrounded Regions (LeetCode 130)

Problem: Flip all ‘O’s that are not connected to the border to ’X’.

Approach: DFS from Border - Perform DFS from all border ‘O’s and mark them as safe - Flip all unvisited ’O’s to ’X’

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

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

void solve(vector<vector<char>>& board) {
    int m = board.size(), n = board[0].size();
    vector<vector<bool>> visited(m, vector<bool>(n, false));

    // Mark all border-connected 'O's as visited
    for(int i = 0; i < m; i++) {
        if(board[i][0] == 'O') dfs(i, 0, board, visited);
        if(board[i][n-1] == 'O') dfs(i, n-1, board, visited);
    }
    for(int j = 0; j < n; j++) {
        if(board[0][j] == 'O') dfs(0, j, board, visited);
        if(board[m-1][j] == 'O') dfs(m-1, j, board, visited);
    }

    // Flip unvisited 'O's to 'X'
    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) {
            if(board[i][j] == 'O' && !visited[i][j]) {
                board[i][j] = 'X';
            }
        }
    }
}
```

3. Number of Enclaves (LeetCode 1020)

Problem: Count number of land cells that cannot reach the border.

Approach: DFS from Border - Perform DFS from all border 1s and mark them as visited - Count remaining unvisited 1s

```
void dfs(int i, int j, vector<vector<int>>& grid,
        vector<vector<bool>>& vis) {
    int m = grid.size(), n = grid[0].size();
    if(i < 0 || i >= m || j < 0 || j >= n ||
       grid[i][j] == 0 || vis[i][j]) return;

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

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

    // Mark all border-connected land as visited
    for(int i = 0; i < m; i++) {
        if(grid[i][0] == 1) dfs(i, 0, grid, visited);
        if(grid[i][n-1] == 1) dfs(i, n-1, grid, visited);
    }
    for(int j = 0; j < n; j++) {
        if(grid[0][j] == 1) dfs(0, j, grid, visited);
        if(grid[m-1][j] == 1) dfs(m-1, j, grid, visited);
    }

    // Count unvisited land cells
    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]) {
                count++;
```

```cpp
                }
            }
        }
        return count;
    }
```

4. Detect Cycle in 2D Grid (LeetCode 1559)

Problem: Detect cycle of same character in grid (length ≥ 4).

Approach: DFS with Parent Tracking

```cpp
bool dfs(pair<int,int> curr, pair<int,int> par,
         vector<vector<char>>& grid,
          vector<vector<bool>>& visited) {
    auto [r, c] = curr;
    visited[r][c] = true;
    int m = grid.size(), n = grid[0].size();
    int dir[] = {1, 0, -1, 0, 1};

    for(int d = 0; d < 4; d++) {
        int nr = r + dir[d], nc = c + dir[d+1];
        if(nr < 0 || nr >= m || nc < 0 || nc >= n ||
           grid[nr][nc] != grid[r][c]) continue;

        if(!visited[nr][nc]) {
            if(dfs({nr, nc}, {r, c}, grid, visited)) return true;
        } else if(make_pair(nr, nc) != par) {
            return true; // Cycle detected
        }
    }
    return false;
}

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

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

Alternative: BFS with Parent Tracking

```
bool containsCycle(vector<vector<char>>& grid) {
    int m = grid.size(), n = grid[0].size();
    vector<vector<bool>> visited(m, vector<bool>(n, false));
    int dir[] = {1, 0, -1, 0, 1};

    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) {
            if(visited[i][j]) continue;

            queue<pair<pair<int,int>, pair<int,int>>> q;
            q.push({{i, j}, {-1, -1}});
            visited[i][j] = true;
            char ch = grid[i][j];

            while(!q.empty()) {
                auto [curr, par] = q.front();
                q.pop();
                int r = curr.first, c = curr.second;

                for(int d = 0; d < 4; d++) {
                    int nr = r + dir[d], nc = c + dir[d+1];
                    if(nr >= 0 && nr < m && nc >= 0 && nc < n &&
                       grid[nr][nc] == ch) {
                        if(!visited[nr][nc]) {
                            visited[nr][nc] = true;
                            q.push({{nr, nc}, {r, c}});
                        } else if(make_pair(nr, nc) != par) {
                            return true;
                        }
                    }
                }
            }
        }
    }
    return false;
}
```

5. Number of Distinct Islands (LeetCode 694)

Problem: Count number of distinct island shapes.

Approach: DFS + Path Encoding - For each unvisited land cell (1), perform DFS - Record the relative coordinates of each cell in the island - Store each unique shape in a set

```
void dfs(int i, int j, int baseRow, int baseCol,
         vector<vector<int>>& grid, vector<vector<bool>>&
        visited,
         vector<pair<int,int>>& shape) {
    int m = grid.size(), n = grid[0].size();
    visited[i][j] = true;
    shape.push_back({i - baseRow, j - baseCol});

    int dir[] = {1, 0, -1, 0, 1};
    for(int d = 0; d < 4; d++) {
        int ni = i + dir[d], nj = j + dir[d+1];
        if(ni >= 0 && ni < m && nj >= 0 && nj < n &&
           grid[ni][nj] == 1 && !visited[ni][nj]) {
            dfs(ni, nj, baseRow, baseCol, grid, visited, shape);
        }
    }
}

int numDistinctIslands(vector<vector<int>>& grid) {
    int m = grid.size(), n = grid[0].size();
    vector<vector<bool>> visited(m, vector<bool>(n, false));
    set<vector<pair<int,int>>> uniqueShapes;

    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) {
            if(grid[i][j] == 1 && !visited[i][j]) {
                vector<pair<int,int>> shape;
                dfs(i, j, i, j, grid, visited, shape);
                uniqueShapes.insert(shape);
            }
        }
    }

    return uniqueShapes.size();
}
```

6. Cycle Detection in Undirected Graph

DFS Approach:

```
bool dfs(int node, int parent, vector<int> adj[], vector<bool>&
        visited) {
    visited[node] = true;
    for(int neighbor : adj[node]) {
        if(!visited[neighbor]) {
            if(dfs(neighbor, node, adj, visited)) return true;
        } else if(neighbor != parent) {
            return true; // Back edge found - cycle detected
        }
    }
    return false;
}

bool hasCycle(int V, vector<int> adj[]) {
    vector<bool> visited(V, false);
    for(int i = 0; i < V; i++) {
        if(!visited[i]) {
            if(dfs(i, -1, adj, visited)) return true;
        }
    }
    return false;
}
```

BFS Approach:

```
bool bfs(int start, vector<int> adj[], vector<bool>& visited) {
    queue<pair<int, int>> q; // {node, parent}
    q.push({start, -1});
    visited[start] = true;

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

        for(int neighbor : adj[node]) {
            if(!visited[neighbor]) {
                visited[neighbor] = true;
                q.push({neighbor, node});
            } else if(neighbor != parent) {
                return true; // Cycle detected
            }
        }
    }
    return false;
}

bool hasCycle(int V, vector<int> adj[]) {
    vector<bool> visited(V, false);
    for(int i = 0; i < V; i++) {
        if(!visited[i]) {
            if(bfs(i, adj, visited)) return true;
        }
    }
    return false;
}
```

Summary Table

Problem	**Best Approach**	**Time Complexity**	**Space Complexity**	**Key Technique**
01 Matrix	Multi-source BFS	O(m×n)	O(m×n)	Simultaneous BFS from all 0s
Surrounded Regions	DFS from Border	O(m×n)	O(m×n)	Mark border-connected regions
Number of Enclaves	DFS from Border	O(m×n)	O(m×n)	Count unreachable land cells
Detect Cycle in Grid	DFS/BFS with Parent	O(m×n)	O(m×n)	Parent tracking to avoid false cycles
Distinct Islands	DFS + Shape Encoding	O(m×n)	O(m×n)	Relative coordinate normalization
		O(V+E)	O(V)	

Problem	Best Approach	Time Complexity	Space Complexity	Key Technique
Cycle in Graph	DFS/BFS with Parent			Back edge detection

Key Insights

1. **Multi-source BFS** is ideal when you need to find shortest distances from multiple sources simultaneously
2. **Border DFS/BFS** is effective for problems involving regions that can/cannot reach boundaries
3. **Parent tracking** is essential for cycle detection to avoid counting the edge you came from
4. **Shape normalization** using relative coordinates allows comparison of geometric patterns
5. **Visited arrays** prevent infinite loops and ensure each cell/node is processed once