694. Number of Distinct Islands

Medium | Depth-First Search | Breadth-First Search | Union Find | Hash Table | Hash Function | Leetcode Link

Problem Description

In this problem, we are tasked to find the number of distinct islands on a given m x n binary grid. Each cell in the grid can either be "1" (representing land) or "0" (representing water). An island is defined as a group of "1"s connected horizontally or vertically. We are also told that all the edges of the grid are surrounded by water, which simplifies the problem by ensuring we don't need to handle edge scenarios where islands might be connected beyond the grid.

To consider two islands as the same, they have to be identical in shape and size, and they must be able to be translated (moved horizontally or vertically, but not rotated or flipped) to match one another. The objective is to return the number of distinct islands—those that are not the same as any other island in the grid.

Intuition

The solution leverages a Depth-First Search (DFS) approach. The core idea is to explore each island, marking the visited 'land' cells, and record the paths taken to traverse each island uniquely. These paths are captured as strings, which allows for easy comparison of island shapes.

Here's the step-by-step intuition behind the approach:

1. Iterate through each cell in the grid.
2. When land ("1") is found, commence a DFS from that cell to visit all connected 'land' cells of that island, effectively marking it to avoid revisiting.
3. While executing DFS, record the direction taken at each step to uniquely represent the path over the island. This is done using a representative numerical code.
4. Append reverse steps at the end of each DFS call to differentiate between paths when the DFS backtracks. This ensures that shapes are uniquely identified even if they take the same paths but in different order.
5. After one complete island is traversed and its path is recorded, add the path string to a set to ensure we only count unique islands.
6. Clear the path and continue the search for the next unvisited 'land' cell to find all islands.
7. After the entire grid is scanned, count the number of unique path strings in the set, which corresponds to the number of distinct islands.

This solution is efficient and elegant because the DFS ensures that each cell is visited only once, and the set data structure automatically handles the uniqueness of the islands.

Solution Approach

The implementation of the solution can be broken down into several key components, using algorithms, data structures, and patterns which are encapsulated in the DFS strategy outlined previously:

1. **Depth-First Search (DFS):**
   - This recursive algorithm is essential for traversing the grid and visiting every possible 'land' cell in an island once. The DFS is initiated whenever a '1' is encountered, and it continues until there are no more adjacent '1's to visit.
2. **Grid Modification to Mark Visited Cells:**
   - As the DFS traverses the grid, it marks the cells that have been visited by flipping them from '1' to '0'. This prevents revisiting the same cell and ensures each 'land' cell is part of only one island.
3. **Directions and Path Encoding:**
   - Four possible directions for movement are captured in a list of deltas dirs. During DFS, the current direction taken is recorded by appending a number to the path list, which is converted to a string for easy differentiation.
4. **Backtracking with Signature:**
   - To handle backtracking uniquely, the algorithm appends a negative value of the move when DFS backtracks. This helps in creating a unique path signature for shapes that otherwise may seem similar if traversed differently.
5. **Path Conversion and Uniqueness:**
   - After a complete island is explored, the path list is converted to a string that represents the full traversal path of the island. This string allows the shape of the island to be expressed uniquely, similar to a sequence of moves in a game.
6. **Set Data Structure:**
   - A set() is used to store unique island paths. This data structure's inherent property to store only unique items simplifies the task of counting distinct islands. Duplicates are automatically handled.
7. **Counts Distinct Islands:**
   - The final number of distinct islands is obtained by measuring the length of the set containing the unique path strings.

Here is the code snippet detailing the DFS logic:

```python
def dfs(i: int, j: int, k: int):
    grid[i][j] = 0
    path.append(str(k))
    dirs = (-1, 0, 1, 0, -1)
    for h in range(1, 5):
        x, y = i + dirs[h - 1], j + dirs[h]
        if 0 <= x < m and 0 <= y < n and grid[x][y]:
            dfs(x, y, h)
    path.append(str(-k))
```

Finally, the number of distinct islands is returned using len(paths) where paths is the set of unique path strings.

By following these stages, the algorithm effectively captures the essence of each island's shape regardless of its location in the grid, allowing us to accurately count the number of distinct islands present.

Example Walkthrough

Let's illustrate the solution approach with a small 4 x 5 grid example:

Assume our grid looks like this, where '1' is land and '0' is water:

```
1 1 0 0 0
1 1 0 1 1
0 0 0 1 1
0 0 0 0 0
```

Now, let's walk through the DFS approach outlined above:

1. Start scanning the grid from (0, 0).
2. At (0, 0), find '1' and start DFS, initializing an empty path list.
3. Visit (0, 0), mark as '0', and add direction code to path. Since there's no previous cell, we skip encoding here.
4. From (0, 0), move to (0, 1) (right), mark as '0', and add '2' to path (assuming '2' represents moving right).
5. Continue DFS to (1, 1) (down), mark as '0', and add '3' to path (assuming '3' represents moving down).
6. DFS can't move further, so backtrack appending '-3' to path to return to (0, 1), then append '-2' to backtrack to (0, 0).
7. Since all adjacent '1's are visited, the DFS for this island ends, and the path is ['2', '3', '-3', '-2'].
8. Convert path to a string "233-3-2" and insert into the paths set.
9. Continue scanning the grid and start a new DFS at (1, 3), where the next '1' is found.
10. Repeat the steps to traverse the second island. Assume the resulting path for the second island is "233-3-2".
11. Conversion and insertion into paths set have no effect as it's a duplicate.
12. Finish scanning the grid, with no more '1's left.
13. Count the distinct islands from the paths set, which contains just one unique path string "233-3-2".

We conclude that there is only 1 distinct island in this example grid.

This walkthrough demonstrates how the DFS exploration with unique path encoding can be used to solve this problem, ensuring that only distinct islands are counted even when they are scattered throughout the grid.

Python Solution

```python
class Solution:
    def numDistinctIslands(self, grid: List[List[int]]) -> int:
        # Depth-first search function to explore the island
        def depth_first_search(i: int, j: int, move: int):
            grid[i][j] = 0  # Marking the visited cell as '0' to avoid revisiting
            path.append(str(move))  # Add the move direction to path
            # Possible movements in the four directions: up, right, down, left
            directions = (-1, 0, 1, 0, -1)
            # Iterating over the four possible directions
            for h in range(4):
                # Calculate new cell's coordinates
                x, y = i + directions[h], j + directions[h+1]
                # Check if the new cell is within bounds and is part of an island
                if 0 <= x < m and 0 <= y < n and grid[x][y]:
                    depth_first_search(x, y, h+1)
            path.append(str(-move))  # Add the reverse move to path to differentiate shapes

        # Set to store unique paths that represent unique island shapes
        paths = set()
        # Temporary list to store the current island's path shape
        path = []
        # Dimensions of the grid
        m, n = len(grid), len(grid[0])

        # Iterate over every cell in the grid
        for i, row in enumerate(grid):
            for j, value in enumerate(row):
                if value:  # Check if the cell is part of an island
                    depth_first_search(i, j, 0)  # Start DFS from this cell
                    paths.add("".join(path))  # Add the path shape to the set of paths
                    path.clear()  # Clear the path for next island

        # Number of distinct islands is the number of unique path shapes
        return len(paths)
```

Java Solution

```java
class Solution {
    private int rows; // number of rows in the grid
    private int cols; // number of columns in the grid
    private int[][] grid; // grid representation
    private StringBuilder path; // used to store the path during DFS to identify unique islands

    public int numDistinctIslands(int[][] grid) {
        rows = grid.length; // set the number of rows
        cols = grid[0].length; // set the number of columns
        this.grid = grid; // reference the grid
        Set<String> uniqueIslands = new HashSet<>(); // store unique island paths as strings
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                if (grid[i][j] == 1) { // if it's part of an island
                    path = new StringBuilder(); // initialize the path
                    exploreIsland(i, j, 'S'); // start DFS with dummy direction 'S' (Start)
                    uniqueIslands.add(path.toString()); // add the path to the set
                }
            }
        }
        return uniqueIslands.size(); // the number of unique islands
    }

    private void exploreIsland(int i, int j, char direction) {
        grid[i][j] = 0; // mark as visited
        path.append(direction); // append the direction to path

        // directions represented as delta x and delta y
        int[] dX = {-1, 0, 1, 0};
        int[] dY = {0, 1, 0, -1};
        char[] dirCodes = {'U', 'R', 'D', 'L'}; // corresponding directional codes

        for (int dir = 0; dir < 4; ++dir) { // iterate over possible directions
            int x = i + dX[dir];
            int y = j + dY[dir];
            if (x >= 0 && x < rows && y >= 0 && y < cols && grid[x][y] == 1) { // check for valid next cell
                exploreIsland(x, y, dirCodes[dir]); // recursive DFS call
            }
        }

        path.append('B'); // append backtrack code to ensure paths are unique after recursion return
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <unordered_set>
#include <functional> // Include necessary headers

class Solution {
public:
    int numDistinctIslands(vector<vector<int>>& grid) {
        unordered_set<string> uniqueIslandPaths; // Store unique representations of islands
        string currentPath; // Store the current traversal path
        int rowCount = grid.size(), colCount = grid[0].size(); // Dimensions of the grid
        int directions[5] = {-1, 0, 1, 0, -1}; // Used for moving in the grid

        // Depth-first search (DFS) to traverse islands and record paths
        function<void(int, int, int)> dfs = [&](int row, int col, int moveDirection) {
            grid[row][col] = 0; // Mark the current cell as visited
            currentPath += to_string(moveDirection); // Record move direction

            // Explore all possible directions: up, right, down, left
            for (int d = 1; d < 5; ++d) {
                int newRow = row + directions[d - 1], newCol = col + directions[d];
                if (newRow >= 0 && newRow < rowCount && newCol >= 0 && newCol < colCount && grid[newRow][newCol]) {
                    // Continue DFS if the next cell is part of the island (i.e., grid value is 1)
                    dfs(newRow, newCol, d);
                }
            }

            // Record move direction again to differentiate between paths with same shape but different sizes
            currentPath += to_string(moveDirection);
        };

        // Iterate over all grid cells to find all islands
        for (int i = 0; i < rowCount; ++i) {
            for (int j = 0; j < colCount; ++j) {
                // If the cell is part of an island
                if (grid[i][j]) {
                    dfs(i, j, 0); // Start DFS
                    uniqueIslandPaths.insert(currentPath); // Add the path to the set
                    currentPath.clear(); // Reset path for the next island
                }
            }
        }

        // The number of unique islands is the size of the set containing unique paths
        return uniqueIslandPaths.size();
    }
};
```

Typescript Solution

```typescript
function numDistinctIslands(grid: number[][]): number {
    const rowCount = grid.length; // The number of rows in the grid.
    const colCount = grid[0].length; // The number of columns in the grid.
    const uniquePaths: Set<string> = new Set(); // Set to store unique island shapes.
    const currentPath: number[] = []; // Array to keep the current DFS path.
    const directions: number[] = [-1, 0, 1, 0, -1]; // Array for row and column movements.

    // Helper function to perform DFS.
    const dfs = (row: number, col: number, directionIndex: number) => {
        grid[row][col] = 0; // Mark the cell as visited by setting it to 0.
        currentPath.push(directionIndex); // Append the direction index to the current path.

        // Explore all four directions: up, right, down, left.
        for (let i = 1; i < 5; ++i) {
            const nextRow = row + directions[i - 1];
            const nextCol = col + directions[i];

            // Check if the next cell is within bounds and not visited.
            if (nextRow >= 0 && nextRow < rowCount && nextCol >= 0 && nextCol < colCount && grid[nextRow][nextCol]) {
                dfs(nextRow, nextCol, i); // Recursively perform DFS on the next cell.
            }
        }

        // Upon return, push the backtracking direction index to the current path.
        currentPath.push(directionIndex);
    };

    // Iterate through all cells of the grid.
    for (let row = 0; row < rowCount; ++row) {
        for (let col = 0; col < colCount; ++col) {
            // If the current cell is part of an island (value is 1).
            if (grid[row][col]) {
                dfs(row, col, 0); // Start DFS from the current cell.
                uniquePaths.add(currentPath.join(',')); // Add the current path to the set of unique paths.
                currentPath.length = 0; // Reset the currentPath for the next island.
            }
        }
    }

    return uniquePaths.size; // Return the number of distinct islands.
}
```

Time and Space Complexity

The time complexity of the provided code is O(M * N), where M is the number of rows and N is the number of columns in the grid. This is due to the fact that we must visit each cell at least once to determine the islands. The dfs function is called for each land cell (1) and it ensures every connected cell is visited only once by marking visited cells as water (0) immediately, thus avoiding revisiting.

The space complexity is O(M * N) in the worst case. This could happen if the grid is filled with land, and a deep recursive call stack is needed to explore the island. Additionally, the path variable which records the shape of each distinct island can in the worst case take up as much space as all the cells in the grid (if there was one very large island), therefore the space complexity would depend on the input size.