1254. Number of Closed Islands

Medium | Depth-First Search | Breadth-First Search | Union Find | Array | Matrix | Leetcode Link

Problem Description

The problem presents a two-dimensional grid which consists of both land (`0`s) and water (`1`s). An island is defined as a group of land cells (or `0`s) that are connected horizontally or vertically (4-directionally). A closed island is an island that is completely surrounded by water (`1`s), meaning it does not touch the edge of the grid at any point.

The goal is to count how many closed islands are present in the grid. For example, a `0` that is at the edge of the grid could be part of an island, but since it is touching the boundary, it cannot be considered a closed island.

Intuition

To solve this problem, we can perform a depth-first search (DFS) on each cell that is land (`0`). When we perform a DFS, we recursively visit each connected land cell and mark them as water (`1`) to avoid revisiting them. While performing the DFS, we also check if we ever touch the boundary of the grid (i.e., the row index `i` is 0 or `m-1` or the column index `j` is 0 or `n-1`). If the DFS starts from a cell that never touches the boundary, it means we are on a closed island, and thus we should count it.

The function `dfs(i, j)` is designed to perform this task, where `i` and `j` are the row and column indices, respectively. This function marks the current cell as visited by setting `grid[i][j]` to `1`. It then checks in all four directions and calls itself (the `dfs` function) on any connected land cell it finds. If any recursive call to `dfs` encounters the boundary, it propagates back a failure to recognize the island as closed by returning `0` (when bitwise `&` operator is applied). If an island is completely surrounded by water and doesn't touch the boundary at any point, `dfs` returns `1`.

The `pairwise(dirs)` function combined with the surrounding `for` loop is a slight mistake in the code, possibly due to a misunderstanding of Python's itertools.pairwise, which doesn’t apply here. Instead, it should loop over pairs of directions (like `(dirs[k], dirs[k + 1])` for `k` in range 4) to check all four neighboring cells.

The final count of closed islands is obtained by initializing a sum with a generator expression that iterates over every cell in the grid and calls the `dfs` function if a land cell (`0`) is found. The result of `dfs` will only be `1` for cells that are part of a closed island, and thus the sum represents the total number of closed islands in the grid.

Note: The code provided seems to have a mistake with the `pairwise(dirs)` portion, which does not align with standard Python functions. The intention here might be to iterate over the directions in pairs to traverse up, right, down, and left in the grid.

Solution Approach

The solution applies the depth-first search (DFS) algorithm to explore the grid and identify islands that are closed. Here's a step-by-step breakdown of the implementation:

1. **Initialization:** A DFS function is defined, which will be used to explore the grid. Additionally, the size of the grid is noted in variables `m` (for rows) and `n` (for columns). The `dirs` array (which should contain pairs of directions to move up, right, down, and left) is used to facilitate the exploration of neighboring cells.

2. **Depth-First Search (DFS):** The core of the solution is in the `dfs` function, which takes a cell `(i, j)` and:

   - Marks it as visited by setting `grid[i][j] = 1`.
   - Checks if the cell is within the boundary but not on it, meaning `0 < i < m - 1` and `0 < j < n - 1`.
   - Iterates in all four directions based on the `dirs` array, and if an adjoining cell is unvisited land (`grid[x][y] == 0`), it recurs with these new coordinates.
   - If the recursive exploration touches the grid boundary, it will return 0, indicating this is not a `closed island`. If the exploration completes without touching the boundary, it returns 1, indicating a `closed island`.
3. **Counting Closed Islands:** The main function now initializes a sum with a generator expression that loops over each cell in the grid. It calls `dfs` on cells that are land (`0`) and adds up the results. Each `1` returned by `dfs` denotes a closed island.

Here, it is worth noting that the "Reference Solution Approach" mentioned Union find. While the given solution does not utilize this pattern, it's relevant to acknowledge that Union find is another approach that can solve this problem. Union find, or disjoint-set data structure, would allow us to group land cells into disjoint sets and then count sets that do not touch the boundary of the grid, which effectively would identify closed islands. However, in the provided solution, DFS is used instead, which is an equally valid method for solving this problem.

The effectiveness of the DFS approach lies in its ability to mark and visit each cell only once, ensuring that the solution is efficient, with the time complexity being O(m*n), where `m` is the number of rows and `n` is the number of columns in the `grid`. The DFS algorithm is a natural fit for problems related to navigating and exploring grids, particularly in searching for connected components, such as islands in this context.

Example Walkthrough

Let's take a small grid as an example to illustrate how the solution approach works. Consider the following grid:

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

In this grid, `1` represents water and `0` represents land. We want to find how many closed islands (areas of `0`s that do not touch the grid's boundary) there are.

1. We start by defining a `dfs` function and initializing the grid size, `m` and `n`, respectively. In this case, `m = 6` (rows) and `n = 6` (columns).

2. We iterate over each cell in the grid. When we come across a land cell (`0`), we start the DFS process from it by calling our `dfs` function.

3. We call the `dfs` function on the land cell in the second row, second column (`grid[1][1]`). The cell is marked as visited by setting it to `1`. DFS checks the neighboring cells in four directions (up, down, left, right) based on a pair of directions.

4. The DFS explores all connected land cells. For this example, it would change the following cells to `1`:

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

5. The DFS does not reach the boundary for this group of land cells, so it counts as one closed island.

6. The next unvisited land cell initiates another DFS call which is on the fourth row, fourth column (`grid[3][3]`). The process is similar; it explores connected land cells:

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

7. However, this time, the DFS touches the boundary at cell (`grid[4][1]`), so this island is not closed.

8. The final unvisited land cell group starts at `grid[4][1]` and is similarly adjacent to the boundary.

At the end of this process, we see that only the first group of land cells (`grid[1][1]`) constitutes a closed island. Thus, the final count is 1.

The DFS algorithm efficiently identified areas that constitute closed islands, ensuring no cell is visited more than once, resulting in a time complexity of O(m*n). While the `pairwise(dirs)` part in the example content is confusing, the logic of the DFS here applies the correct four-directional checks by iterating over each direction pair to look at neighboring cells.

Python Solution

```python
class Solution:
    def closedIsland(self, grid: List[List[int]]) -> int:
        # Perform depth-first search to find and mark all cells connected to (i, j).
        def dfs(row: int, col: int) -> int:
            # Check if the cell is within the inner part of the grid
            # Considering the edges can't form a closed island.
            result = 1 if 0 < row < rows - 1 and 0 < col < cols - 1 else 0

            # Mark the current cell as visited
            grid[row][col] = 1

            # Explore all four directions (up, right, down, left)
            for direction in range(4):
                new_row = row + directions[direction]
                new_col = col + directions[direction + 1]

                # If the new cell is within the grid and not visited
                if 0 <= new_row < rows and 0 <= new_col < cols and grid[new_row][new_col] == 0:
                    # Continue the DFS and use logical AND to ensure all connected cells are closed
                    result &= dfs(new_row, new_col)
            return result

        # Get the number of rows and columns in the grid
        rows, cols = len(grid), len(grid[0])

        # Define the directions for the DFS. The pairs are (up, right, down, left).
        directions = (-1, 0, 1, 0, -1)

        # Initialize the count of closed islands
        closed_islands = 0

        # Iterate over each cell of the grid
        for i in range(rows):
            for j in range(cols):
                # If the cell is land (0) and has not been visited
                if grid[i][j] == 0:
                    # Perform DFS from the current cell and check if it forms a closed island
                    closed_islands += dfs(i, j)

        # Return the total count of closed islands
        return closed_islands

```

Java Solution

```java
class Solution {
    private int height;  // Variable to store the height of the grid
    private int width;   // Variable to store the width of the grid
    private int[][] grid; // 2D array to represent the grid itself

    // Function to count the number of closed islands in the grid
    public int closedIsland(int[][] grid) {
        height = grid.length;    // Set the height of the grid
        width = grid[0].length;  // Set the width of the grid
        this.grid = grid;        // Assign the grid argument to the instance variable
        int count = 0;           // Initialize count of closed islands

        // Iterate over each cell in the grid
        for (int i = 0; i < height; ++i) {
            for (int j = 0; j < width; ++j) {
                // If the current cell is 0 (land), perform a DFS to mark the connected components
                if (grid[i][j] == 0) {
                    count += dfs(i, j); // Increment count if the land is part of a closed island
                }
            }
        }
        return count; // Return the total number of closed islands
    }

    // Helper method to perform DFS and determine if a connected component is a closed island
    private int dfs(int row, int col) {
        // If not on the border, assume it's a closed island; otherwise, it's not
        int isClosed = (row > 0 && row < height - 1 && col > 0 && col < width - 1) ? 1 : 0;
        grid[row][col] = 1; // Mark the current cell as visited by setting it to 1
        int[] directions = {-1, 0, 1, 0, -1}; // Array to help traverse in 4 cardinal directions

        // Explore all 4 adjacent cells
        for (int k = 0; k < 4; ++k) {
            int nextRow = row + directions[k];          // Calculate the next row index
            int nextCol = col + directions[k + 1];      // Calculate the next column index
            // Verify if the adjacent cell is within the grid and if it is land (0)
            if (nextRow >= 0 && nextRow < height && nextCol >= 0 && nextCol < width
                && grid[nextRow][nextCol] == 0) {
                // Perform DFS on the adjacent cell and use logical AND to update the isClosed status
                isClosed &= dfs(nextRow, nextCol); // If any part touches the border, it's not closed
            }
        }
        return isClosed; // Return 1 if the island is closed, otherwise 0
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int closedIsland(vector<vector<int>>& grid) {
        // Get the number of rows and columns in the grid.
        int rowCount = grid.size(), colCount = grid[0].size();

        // Initialize the count of closed islands to zero.
        int closedIslandCount = 0;

        // Define directions for traversing the grid (up, right, down, left).
        int directions[5] = {-1, 0, 1, 0, -1};

        // Define a recursive Depth-First Search (DFS) lambda function to find closed islands.
        function<int(int, int)> depthFirstSearch = [&](int row, int col) -> int {
            // If the current cell is surrounded by non-border cells, mark as potential closed island.
            int isClosedIsland = (row > 0 && row < rowCount - 1 && col > 0 && col < colCount - 1) ? 1 : 0;

            // Mark the current cell as visited.
            grid[row][col] = 1;

            // Explore all 4 directions.
            for (int k = 0; k < 4; ++k) {
                int nextRow = row + directions[k], nextCol = col + directions[k + 1];
                // If the next cell is on the grid and unvisited, continue the DFS.
                if (nextRow >= 0 && nextRow < rowCount && nextCol >= 0 && nextCol < colCount && grid[nextRow][nextCol] == 0) {
                    // If any recursive DFS call returns 0, then it's not a closed island.
                    isClosedIsland &= depthFirstSearch(nextRow, nextCol);
                }
            }
            // Return whether the current cell is part of a closed island.
            return isClosedIsland;
        };

        // Loop over all cells in the grid.
        for (int i = 0; i < rowCount; ++i) {
            for (int j = 0; j < colCount; ++j) {
                // If the cell is land (0) and the DFS returns true for a closed island, increment the count.
                closedIslandCount += (grid[i][j] == 0 && depthFirstSearch(i, j));
            }
        }

        // Return the total count of closed islands found.
        return closedIslandCount;
    }
};

```

Typescript Solution

```typescript
// Function to calculate the number of closed islands in a grid.
// A closed island is a group of connected 0s that is completely surrounded by 1s (representing water),
// and does not touch the edge of the grid.
// @param grid - The 2D grid of 0s (land) and 1s (water).
// @returns The count of closed islands.
function closedIsland(grid: number[][]): number {
    // Store the dimensions of the grid.
    const rowCount = grid.length;
    const colCount = grid[0].length;
    // Directions for traversing up, down, left, and right in the grid.
    const directions = [-1, 0, 1, 0, -1];
    // DFS function to explore the cells in the grid.
    const depthFirstSearch = (row: number, col: number): number => {
        // A flag to indicate if the current island is closed.
        // It's closed only if it doesn't touch the grid's edge.
        let isClosed = row > 0 && col > 0 && row < rowCount - 1 && col < colCount - 1 ? 1 : 0;
        // Mark the cell as visited by setting it to 1.
        grid[row][col] = 1;
        // Explore all 4 adjacent directions (up, down, left, right).
        for (let k = 0; k < 4; ++k) {
            const newRow = row + directions[k];
            const newCol = col + directions[k + 1];
            // Check if the new cell is within bounds and is land (0).
            if (newRow >= 0 && newCol >= 0 && newRow < rowCount && newCol < colCount && grid[newRow][newCol] === 0) {
                // If any recursive call returns 0, the island isn't closed.
                isClosed &= depthFirstSearch(newRow, newCol);
            }
        }
        // Return whether the island is closed.
        return isClosed;
    };

    // Initialize the count of closed islands.
    let closedIslandsCount = 0;
    // Iterate through all cells in the grid.
    for (let row = 0; row < rowCount; ++row) {
        for (let col = 0; col < colCount; col++) {
            // If a cell is land and unvisited, perform DFS.
            if (grid[row][col] === 0) {
                closedIslandsCount += depthFirstSearch(row, col);
            }
        }
    }
    // Return the final count of closed islands.
    return closedIslandsCount;
}

```

Time and Space Complexity

The given Python code is a solution for finding the number of "closed islands" in a 2D grid, where `0` represents land and `1` represents water. A "closed island" is one that is surrounded by water and doesn't touch the edge of the grid. The algorithm uses Depth-First Search (DFS) to explore the grid.

Time Complexity

The time complexity of the code is `O(m * n)`, where `m` is the number of rows and `n` is the number of columns in the grid. This is because the algorithm potentially visits each cell in the grid once. The DFS is started only if the current cell is a `0` (land), and then it marks every visited land cell as `1` (water) to prevent revisiting. In the worst case, all cells in the grid will be traversed once.

Space Complexity

The space complexity of the DFS is `O(m * n)` in the worst case. This might happen when the depth of the recursion stack grows proportionally to the number of cells in the worst-case scenario, such as when the grid forms a large area of contiguous land. The space is used by the recursion call stack during the depth-first search.

However, if we consider that modern Python implementations use a technique called "tail-recursion elimination" in some cases, the space complexity can be effectively less than `O(m * n)` for some grids because not all calls will result in an additional stack frame. But in the analysis of DFS, it is conventional to consider the worst-case scenario regarding recursion.

The additional space used by the `dirs` variable and a few integer variables is negligible compared to the recursive stack space, thus not affecting the overall space complexity.