1162. As Far from Land as Possible

Medium | Breadth-First Search | Array | Dynamic Programming | Matrix

Problem Description

In this problem, we are given a two-dimensional array `grid` of size `n x n` that represents a map of water and land. The values in the grid can either be 0 or 1. A value of 0 indicates a water cell, while a value of 1 indicates a land cell. Our goal is to find a water cell such that it is as far away as possible from any land cell. This is measured by the Manhattan distance, which is the sum of the absolute differences of their coordinates. In other words, the Manhattan distance between two cells `(x0, y0)` and `(x1, y1)` is `|x0 - x1| + |y0 - y1|`. If the grid contains only land or only water, we are to return -1. Essentially, we are looking for the maximum distance from water to land, and this distance needs to be the shortest path, as calculated by the Manhattan distance.

Intuition

The problem can be approached by using a Breadth-First Search (BFS) algorithm starting from all the land cells simultaneously. This way, we can calculate the distance of each water cell from the nearest land cell. The BFS algorithm works well here because it naturally explores elements level by level, starting with nodes that are closest (in terms of edge count) to the starting nodes. By starting from every land cell, we ensure that the first time we reach a water cell is the shortest distance to land because BFS doesn't revisit nodes that it has already seen.

To implement this, we first create a queue and enqueue all the land cell coordinates. Then, we iterate over the queue in a BFS manner. For each cell that we pop from the queue, we check all of its adjacent (up, down, left, right) cells. If an adjacent cell is water (indicated by a 0), we mark it as visited by changing its value to 1 and then add it to the queue. This process continues until there are no more water cells left to visit.

We keep track of the levels in BFS because each level represents an additional step away from the originally enqueued land cells. Once BFS completes, the last level's distance is the furthest any water cell is from the nearest land cell (the maximum distance). If there are no water cells, or the grid is entirely water, the answer is -1, as per the problem statement.

In the provided solution, the Python function `pairwise` is used to generate adjacent direction pairs (up, down, left, right) from the `dirs` list. This is a convenience for traversing the grid.

Solution Approach

The solution uses the Breadth-First Search (BFS) algorithm to efficiently find the maximum distance of a water cell to the nearest land cell. Here is a step-by-step breakdown of how the solution is implemented:

1. A `deque`, which is a double-ended queue from the collections module in Python, is used because it supports constant time append and pop operations from both ends. This is ideal for a BFS implementation where we need to efficiently add to and remove elements from the queue.
2. The first step is to locate all the land cells and add their coordinates to the queue. This is done with the list comprehension `q = deque((i, j) for i in range(n) for j in range(n) if grid[i][j])`, which scans the entire grid and enqueues the coordinates of all cells with a value of 1 (land).
3. In the case where the grid is all land or all water, the function immediately returns `-1`, since it's not possible to have a distance from water to land in these scenarios.
4. The solution introduces a list `dirs` which holds values `(-1, 0, 1, 0, -1)`. With the `pairwise` function, each pair from this list will be used to represent the directional vectors (up, down, left, right) for the adjacent cells.
5. As long as the queue is not empty, the BFS process continues. For each cell in the queue, we remove it from the front of the queue, and for each direction (up, down, left, right), we compute the neighboring cell's coordinates.
6. If a neighbor cell is within the bounds of the grid and is water (indicated by a 0), we mark it as visited (by marking it as land with a 1) to prevent re-visitation and enqueue it.
7. After visiting all neighbors of all cells in the current level of the BFS, we increment a counter `ans`. This counter keeps track of the distance from the initial set of land cells. Each level of the BFS represents moving one step further out from the land cells, so when the last level is reached, `ans` represents the maximum distance.

The code utilizes a BFS to ensure that we always find the shortest path from any water cell to land because BFS guarantees that the shortest path will be found first before any longer paths. By visiting cells level-by-level and marking visited cells, the algorithm efficiently expands outwards from all pieces of land simultaneously. The final result is the maximum distance to the nearest land for any water cell in the grid.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using a 3×3 grid. Consider the following grid:

```
grid = [
    [0, 0, 1],
    [0, 1, 0],
    [1, 0, 0]
]
```

Here, `1` denotes land and `0` denotes water.

Step 1: Initialize an empty `deque` and add all land cells' coordinates to it. Our grid has land cells at coordinates (0,2), (1,1), and (2,0), so the `deque` will look like this: `deque([(0, 2), (1, 1), (2, 0)])`.

Step 2: Since the grid contains both land and water, we do not return -1.

Step 3: We use the list `dirs = (-1, 0, 1, 0, -1)` to help us find adjacent cells.

Step 4: Begin the BFS process. We dequeue (pop from the front) a cell, check its unvisited (0 valued) neighbors, mark them as visited (1), and enqueue their coordinates.

Here's how the BFS process progresses:

- For (0, 2), its unvisited neighbors are (0, 1) and (1, 2). We visit them, mark them as land, and enqueue their coordinates. The grid becomes:

```
grid = [
    [0, 1, 1],
    [0, 1, 1],
    [1, 0, 0]
]
```

The `deque` now becomes: `deque([(1, 1), (2, 0), (0, 1), (1, 2)])`.

- Next, for (1, 1), all neighbors are already visited or are itself, so no new cells are enqueued.
- Then, for (2, 0), its unvisited neighbor is (2, 1). We visit it, mark it, and enqueue it. The grid becomes:

```
grid = [
    [0, 1, 1],
    [0, 1, 1],
    [1, 1, 0]
]
```

The `deque` now becomes: `deque([(0, 1), (1, 2), (2, 1)])`.

- Continue the process for each cell in the `deque`, then enqueue their unvisited neighbors, marking them as visited, and the next level of BFS begins.
- After processing all the cells in the queue, with no more water cells left to visit, the algorithm stops. We increment `ans` each time we go deeper into a new level in the queue.

At the end of BFS, before the last water cell at (2, 2) is marked as land, `ans = 2`, which is the Manhattan distance from this water cell to the nearest land cell. Since this was the last water cell visited, the function would return 2. Note that in the real implementation, we would keep track of whether a cell has been visited differently to conserve the grid's structure and information after finishing the algorithm.

Python Solution

```python
from collections import deque

class Solution:
    def maxDistance(self, grid: List[List[int]]) -> int:
        # Get the size of the grid
        n = len(grid)

        # Initialize a queue with all land cells (cells with value 1)
        queue = deque((i, j) for i in range(n) for j in range(n) if grid[i][j])

        # Initialize the maximum distance as -1
        max_distance = -1

        # If the grid is all water or all land, return -1 as per problem statement
        if len(queue) in (0, n * n):
            return max_distance

        # Directions for exploring adjacent cells (up, right, down, left)
        directions = ((-1,0), (0,1), (1,0), (0,-1))

        # Begin BFS from all land cells
        while queue:
            # Increment distance for each level (layer) of BFS
            max_distance += 1
            for _ in range(len(queue)):
                i, j = queue.popleft()

                # Explore all neighboring cells
                for di, dj in directions:
                    x, y = i + di, j + dj

                    # Check if the neighboring cell is within the grid and is water
                    if 0 <= x < n and 0 <= y < n and grid[x][y] == 0:
                        # Mark the water cell as visited (converted to land)
                        grid[x][y] = 1
                        # Add the water cell to the queue to continue BFS
                        queue.append((x, y))

        # Return the maximum distance from any land cell to the furthest water cell
        return max_distance

```

Java Solution

```java
class Solution {
    public int maxDistance(int[][] grid) {
        int gridSize = grid.length; // The size of the grid
        Deque<int[]> queue = new ArrayDeque<>(); // Queue to facilitate BFS

        // Traverse the grid to find all land cells (value 1) and add them to the queue
        for (int i = 0; i < gridSize; ++i) {
            for (int j = 0; j < gridSize; ++j) {
                if (grid[i][j] == 1) {
                    queue.offer(new int[] {i, j});
                }
            }
        }

        // Initialize max distance as -1
        int maxDistance = -1;
        // If there are no lands or the entire grid is land, return -1
        if (queue.isEmpty() || queue.size() == gridSize * gridSize) {
            return maxDistance;
        }

        // Array to help calculate the 4-neighbor coordinates
        int[] directions = {-1, 0, 1, 0, -1}; // up, right, down, left

        // Breadth-First Search (BFS)
        while (!queue.isEmpty()) {
            int currentSize = queue.size();
            for (int i = currentSize; i > 0; --i) {
                int[] point = queue.poll();

                // Explore the neighbors of the current cell
                for (int k = 0; k < 4; ++k) { // 4 because there are 4 possible directions
                    int x = point[0] + directions[k];
                    int y = point[1] + directions[k + 1];

                    // Check bounds and whether the cell is water
                    if (x >= 0 && x < gridSize && y >= 0 && y < gridSize && grid[x][y] == 0) {
                        grid[x][y] = 1; // Mark the water cell as visited by converting it to land
                        queue.offer(new int[] {x, y});
                    }
                }
            }
            // Increment the distance after finishing one level of BFS
            ++maxDistance;
        }
        // Return the final calculated max distance
        return maxDistance;
    }
}

```

C++ Solution

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

class Solution {
public:
    int maxDistance(vector<vector<int>>& grid) {
        int gridSize = grid.size(); // Grid size represents the dimension of the grid.
        queue<pair<int, int>> landCells; // Queue to keep track of all land cells.

        // Iterate through the grid and add all land cells to the queue.
        for (int row = 0; row < gridSize; ++row) {
            for (int col = 0; col < gridSize; ++col) {
                if (grid[row][col] == 1) {
                    landCells.emplace(row, col); // Enqueue the land cell's coordinates.
                }
            }
        }

        // Initialize the answer to -1, which signifies that we haven't started exploring yet.
        int maxDist = -1;

        // If there are no land cells or the grid is all land, return -1 since the maxDistance cannot be calculated.
        if (landCells.empty() || landCells.size() == gridSize * gridSize) {
            return maxDist;
        }

        // Define directions for exploring adjacent cells.
        int directions[5] = {-1, 0, 1, 0, -1};

        // Perform a BFS to find the maximum distance from the land to the water.
        while (!landCells.empty()) {
            // Process all cells at the current level.
            for (int i = landCells.size(); i > 0; --i) {
                // Retrieve the front cell's coordinates.
                auto [row, col] = landCells.front();
                landCells.pop(); // Remove the current cell from the queue.

                // Explore all possible adjacent directions.
                for (int k = 0; k < 4; ++k) {
                    int newRow = row + directions[k];
                    int newCol = col + directions[k + 1];

                    // Check if the new cell is within the grid bounds and is a water cell.
                    if (newRow >= 0 && newRow < gridSize && newCol >= 0 && newCol < gridSize && grid[newRow][newCol] == 0) {
                        grid[newRow][newCol] = 1; // Mark the new cell as visited by setting it to land.
                        landCells.emplace(newRow, newCol); // Add new cell to the queue for the next level of BFS.
                    }
                }
            }
            // Increase the maximum distance after completing the current level.
            ++maxDist;
        }

        return maxDist; // Return the maximum distance after BFS traversal.
    }
};

```

Typescript Solution

```typescript
function maxDistance(grid: number[][]): number {
    // n represents the size of the grid.
    const gridSize = grid.length;
    // queue will store the coordinates of cells with value 1.
    const queue: [number, number][] = [];

    // Enqueue all cells with value 1.
    for (let i = 0; i < gridSize; ++i) {
        for (let j = 0; j < gridSize; ++j) {
            if (grid[i][j] === 1) {
                queue.push([i, j]);
            }
        }
    }

    // Initialize the answer to -1, as a flag for no valid answer cases.
    let answer = -1;
    // If the queue is empty or if it's filled with all grid cells, return -1.
    if (queue.length === 0 || queue.length === gridSize * gridSize) {
        return answer;
    }

    // Array for getting the 4 adjacent directions (up, right, down, left) easily.
    const directions: number[] = [-1, 0, 1, 0, -1];

    // BFS to find maximum distance.
    while (queue.length > 0) {
        // Iterate over all cells at current distance.
        for (let count = queue.length; count; --count) {
            // Dequeue the front cell.
            const [currentRow, currentCol] = queue.shift()!;

            // Try all 4 directions.
            for (let k = 0; k < 4; ++k) {
                const newRow = currentRow + directions[k];
                const newCol = currentCol + directions[k + 1];

                // Check if the new position is within the grid bounds and has value 0.
                if (newRow >= 0 && newRow < gridSize && newCol >= 0 && newCol < gridSize && grid[newRow][newCol] === 0) {
                    // Update the grid and enqueue the new position.
                    grid[newRow][newCol] = 1;
                    queue.push([newRow, newCol]);
                }
            }
        }
        // Increment the distance for next round of BFS.
        ++answer;
    }

    // Return the maximum distance found.
    return answer;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is primarily determined by the while loop that processes each cell of the grid. Assuming the grid is an `n x n` matrix, the BFS (Breadth-First Search) process will visit each cell at most once. Initially, all the land cells (cells with a value of `1`) are added to the queue, and for each land cell, the code iterates over its four neighboring cells. In the worst-case scenario, the queue could contain all the cells of the grid (but not at the same time since the grid gets populated with `1`s as the BFS progresses). Hence, the worst-case time complexity is `O(n^2)`, as each cell is processed at most once.

Space Complexity

The space complexity of the code is mainly due to the queue (`q`) that is used for BFS and the `grid` itself. The size of the queue can grow up to `O(n^2)` in a scenario where all cells have to be added before reaching any water cells (cells with a value of `0`) because the queue could contain all the cells in the case of a full traversal. However, it is important to note that the queue will not hold all `n^2` cells at the same time since cells are continuously being removed and added. Nonetheless, the upper bound for the queue's size is still `O(n^2)`. The `grid` is also `O(n^2)` in size, but since it is given as an input and is modified in place, it may not count towards additional space complexity in some analysis models.

Considering only auxiliary space used by the algorithm, and not the input size, the space complexity for the queue would thus be `O(n^2)` as well, which is the additional space required beyond the input itself.