1765. Map of Highest Peak

Medium Breadth-First Search Array Matrix Leetcode Link

Problem Description

Given a matrix `isWater` representing a map with land and water cells, the task is to assign each cell a height based on certain rules. Cells with `isWater[i][j] == 0` are land, and cells with `isWater[i][j] == 1` are water. Water cells must have a height of `0`, while land cells must be assigned a non-negative height. The key requirement is that any pair of adjacent cells must differ in height by at most `1`. The goal is to maximize the highest height in the matrix while following the aforementioned conditions. The solution should return a new matrix, `height`, where `height[i][j]` represents the height of cell `(i, j)`.

Intuition

The problem is similar to filling a terrain with water and watching the water rise, respecting the rule that water doesn't spill over to neighboring cells if the height difference is greater than 1. A good way to approach this problem is by using Multi-Source Breadth-First Search (BFS), which starts from all the water cells simultaneously since we know their height is 0. We consider each water cell as a source and begin to explore outward to adjacent land cells, increasing the height step by step.

As multiple water sources spread out, the "wavefront" of land cells being assigned heights grows. Each step from a water cell is assigning a height of 1 to adjacent land cells, and as we continue BFS, we move from these land cells to their adjacent cells, incrementing height by 1 each time. This process naturally ensures that land cells near water are lower, and as we move away, the height increases up to the point where the maximum height that follows the rules is achieved.

The key to BFS is to visit each cell only once and assign the correct height the first time it is visited, which avoids the need for recalculation. The BFS process continues until there are no more cells to update, meaning every cell has been assigned a height as per the rule.

Solution Approach

The solution utilizes the Multi-Source Breadth-First Search (BFS) algorithm to iterate through the matrix. Let's walk through the key components and the algorithm's implementation:

1. **Initialization**: We create a new matrix `ans` with the same dimensions as `isWater`, initializing all cells with `-1`. This will eventually store the final heights of all cells. Then, we declare a queue `q` to manage the BFS process.
2. **Discovering Water Sources**: We iterate through `isWater`, and for each cell that contains water, we set the corresponding cell in `ans` to `0` (since the height of water cells is always `0`) and add the cell coordinates to `q`.
3. **BFS Process**: We start the BFS by popping elements from the queue `q`. For each cell `(i, j)`, we consider its adjacent cells in the four cardinal directions (north, south, east, west) - these are the potential "next steps" in the BFS. To check for valid adjacent cells, we ensure they are within the matrix bounds and that their height has not been set yet (indicated by `-1`).
4. **Updating Heights**: If an adjacent cell is a valid next step, we set its height to be one more than the current cell's `(i, j)` height. This step ensures that the height difference condition is satisfied. Doing this for all water source cells gradually propagates through the matrix.
5. **Queueing Next Cells**: After updating the height of an adjacent cell, we add the cell coordinates to the queue `q`. Placing the cell in the queue ensures that its adjacent cells will be visited in subsequent iterations of BFS.
6. **Repeating Process**: The BFS continues until current passes through the entire matrix, visiting each cell and assigning the correct height. When the queue is empty, every cell has been reached, and their heights satisfy the rules provided.
7. **Returning Results**: When there are no more cells to update, the BFS is completed, and we return the `ans` matrix, which now contains the heights of all cells.

By converging from multiple sources and using a queue for BFS, the solution efficiently spreads out to maximize the heights in the matrix. It solely relies on simple iterations and checks, ensuring it meets the problem constraints effectively.

Here is the code snippet that embodies the approach:

```python
class Solution:
    def highestPeak(self, isWater: List[List[int]]) -> List[List[int]]:
        m, n = len(isWater), len(isWater[0])
        ans = [[-1] * n for _ in range(m)]
        q = deque()
        for i, row in enumerate(isWater):
            for j, v in enumerate(row):
                if v:
                    q.append((i, j))
                    ans[i][j] = 0
        while q:
            i, j = q.popleft()
            for a, b in pairwise((-1, 0, 1, 0, -1)):
                x, y = i + a, j + b
                if 0 <= x < m and 0 <= y < n and ans[x][y] == -1:
                    ans[x][y] = ans[i][j] + 1
                    q.append((x, y))
        return ans
```

Example Walkthrough

Let's consider a small example of the `isWater` matrix to see how the solution works step by step:

```
Given 'isWater' matrix:
[0, 1]
[0, 0]
```

We have a 2×2 matrix where the top right cell represents water (`1`) and the rest are land (`0`).

1. **Initialization**: We initialize the `ans` matrix with all `-1` to represent unset heights.

```
'ans' matrix after initialization:
[-1, -1]
[-1, -1]
```

2. **Discovering Water Sources**: We find that `isWater[0][1]` is a water source, so we set the corresponding cell in `ans` to `0` and add this cell to our BFS queue `q`.

```
'ans' matrix after discovering water sources:
[-1,  0]
[-1, -1]

Queue 'q' contains:
[(0, 1)]
```

3. **BFS Process**: Dequeue (0, 1) from `q` and look for adjacent land cells.
4. **Updating Heights**: We find two adjacent cells to (0, 1) that are `[0][0]` and `[1][1]`. They are both valid moves, so we set their heights to `1`. Then we enqueue these cells.

```
'ans' matrix after the first BFS iteration:
[ 1,  0]
[-1, -1]

Queue 'q' contains:
[(0, 0), (1, 1)]
```

5. **Queueing Next Cells**: We remove (0, 0) from the queue. The adjacent land cells `[1][0]` can be updated to `2`. Now we enqueue `(1, 0)`.

```
'ans' matrix after updating (0, 0):
[ 1,  0]
[ 2, -1]

Queue 'q' contains:
[(1, 1), (1, 0)]
```

6. **Repeating Process**: We dequeue `(1, 1)`. Since it is already at the boundary, there are no new cells to consider.

Then we dequeue `(1, 0)` and check its adjacents but there are no `-1` values in `ans` meaning all land cells have been reached and heights assigned.

```
'ans' matrix after BFS is completed:
[ 1,  0]
[ 2,  1]

Queue 'q' is empty.
```

7. **Returning Results**: Since the BFS queue is empty, we completed the BFS of all cells, and the `ans` matrix is returned with heights satisfying the rule that adjacent cells differ by at most 1.

The final `ans` matrix is:

```
[ 1,  0]
[ 2,  1]
```

This walkthrough demonstrates how the process of Multi-Source Breadth-First Search works to assign heights to cells while adhering to the provided constraints and maximizing the heights of land cells.

Python Solution

```python
from collections import deque

class Solution:
    def highestPeak(self, is_water: List[List[int]]) -> List[List[int]]:
        # Retrive the number of rows (m) and columns (n)
        num_rows, num_cols = len(is_water), len(is_water[0])

        # Initialize answer grid with -1s indicating unvisited cells
        answer_grid = [[-1] * num_cols for _ in range(num_rows)]
        # Queue to hold the cells to visit in Breadth-First Search (BFS)
        queue = deque()

        # Populate the queue with water cells and set their height to 0 in answer_grid
        for row_index, row in enumerate(is_water):
            for col_index, value in enumerate(row):
                if value:
                    queue.append((row_index, col_index))
                    answer_grid[row_index][col_index] = 0

        # Directons for exploring neighboring cells (up, right, down, left)
        directions = [(-1, 0), (0, 1), (1, 0), (0, -1)]

        # Perform BFS to populate the answer grid
        while queue:
            # Dequeue the first cell
            current_row, current_col = queue.popleft()

            # Explore all neighboring cells
            for delta_row, delta_col in directions:
                # Calculate neighbor coordinates
                neighbor_row, neighbor_col = current_row + delta_row, current_col + delta_col
                # Check if neighbor is within the grid and is unvisited
                if (0 <= neighbor_row < num_rows and
                        0 <= neighbor_col < num_cols and
                        answer_grid[neighbor_row][neighbor_col] == -1):
                    # Update the height of the neighbor cell
                    answer_grid[neighbor_row][neighbor_col] = answer_grid[current_row][current_col] + 1
                    # Enqueue the neighbor cell for visiting
                    queue.append((neighbor_row, neighbor_col))

        # Return the populated answer grid
        return answer_grid

```

Java Solution

```java
class Solution {
    public int[][] highestPeak(int[][] isWater) {
        // Obtain dimensions of the input matrix.
        int rows = isWater.length;
        int cols = isWater[0].length;

        // Initialize the answer matrix with the same dimensions.
        int[][] highestPeaks = new int[rows][cols];

        // Queue for BFS (Breadth-first search).
        Deque<int[]> queue = new ArrayDeque<>();

        // Initialize answer matrix and enqueue all water cells.
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // Mark water cells with 0 and land cells with -1
                highestPeaks[i][j] = isWater[i][j] - 1;

                // Add water cell coordinates to the queue.
                if (highestPeaks[i][j] == 0) {
                    queue.offer(new int[] {i, j});
                }
            }
        }

        // Directions for exploring adjacent cells (up, right, down, left).
        int[] directions = {-1, 0, 1, 0, -1};

        // Perform BFS to find the highest peak values.
        while (!queue.isEmpty()) {
            // Poll a cell from the queue.
            int[] position = queue.poll();
            int row = position[0];
            int col = position[1];

            // Explore all adjacent cells.
            for (int k = 0; k < 4; ++k) {
                // Calculate coordinates of the adjacent cell.
                int x = row + directions[k];
                int y = col + directions[k + 1];

                // Check if the adjacent cell is within bounds and if it is land.
                if (x >= 0 && x < rows && y >= 0 && y < cols && highestPeaks[x][y] == -1) {
                    // Set the height of the land cell to be one more than the current cell.
                    highestPeaks[x][y] = highestPeaks[row][col] + 1;

                    // Enqueue the position of the land cell.
                    queue.offer(new int[] {x, y});
                }
            }
        }

        // Return the filled highestPeaks matrix.
        return highestPeaks;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Define directions for exploring adjacent cells: up, right, down, left.
    const vector<int> dirs{-1, 0, 1, 0, -1};

    // Function to calculate the highest peak elevation for each cell.
    vector<vector<int>> highestPeak(vector<vector<int>>& isWater) {
        int m = isWater.size();      // Number of rows.
        int n = isWater[0].size();   // Number of columns.
        vector<vector<int>> ans(m, vector<int>(n));
        queue<pair<int, int>> q;     // Queue to perform a breadth-first search.

        // Initialize the answer matrix and enqueue all water cells.
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                ans[i][j] = isWater[i][j] - 1; // Water cells become 0, land cells -1.
                if (ans[i][j] == 0) {
                    // If the cell is water, add it to the queue for BFS.
                    q.emplace(i, j);
                }
            }
        }

        // Perform breadth-first search to determine elevation of each cell.
        while (!q.empty()) {
            auto [i, j] = q.front(); // Get the front cell from the queue.
            q.pop();
            for (int k = 0; k < 4; ++k) { // Explore all 4 adjacent cells.
                int x = i + dirs[k];    // Calculate the new x-coordinate.
                int y = j + dirs[k + 1];// Calculate the new y-coordinate.
                // Check if the new coordinates are in bounds and the cell is not yet visited.
                if (x >= 0 && x < m && y >= 0 && y < n && ans[x][y] == -1) {
                    // Set the elevation of the cell to be one more than current cell's elevation.
                    ans[x][y] = ans[i][j] + 1;
                    // Add this cell to the queue to continue BFS from here.
                    q.emplace(x, y);
                }
            }
        }
        return ans; // Return the elevation matrix.
    }
};

```

Typescript Solution

```typescript
function highestPeak(isWater: number[][]): number[][] {
    const rows = isWater.length;    // Number of rows
    const cols = isWater[0].length; // Number of columns
    let answer: number[][] = [];    // Matrix to store the final height map
    let queue: number[][] = [];     // Queue to perform Breadth-First Search (BFS)

    // Initialize the answer matrix with -1 and queue with the water cells (0 elevation)
    for (let i = 0; i < rows; i++) {
        answer.push(new Array(cols).fill(-1));
        for (let j = 0; j < cols; j++) {
            if (isWater[i][j] === 1) {
                queue.push([i, j]);
                answer[i][j] = 0;
            }
        }
    }

    // Directions for exploring the neighbors of a cell (up, right, down, left)
    const directions = [-1, 0, 1, 0, -1];

    // Perform BFS to populate the answer matrix with height/elevation levels
    while (queue.length) {
        let tempQueue: number[][] = [];
        for (const [currentRow, currentCol] of queue) {
            for (let k = 0; k < 4; k++) { // Explore the 4 neighbors of the current cell
                const newRow = currentRow + directions[k];
                const newCol = currentCol + directions[k + 1];
                // If the new cell is within bounds and not yet visited (elevation -1)
                if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && answer[newRow][newCol] === -1) {
                    tempQueue.push([newRow, newCol]);
                    answer[newRow][newCol] = answer[currentRow][currentCol] + 1;
                }
            }
        }
        queue = tempQueue; // Update the queue with the next level of cells to be visited
    }

    return answer; // Return the populated height map
}

```

Time and Space 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 matrix `isWater`. This is because the algorithm must visit every cell at least once to determine its height in the final output matrix, and in the worst case, each cell is enqueued and dequeued exactly once in the BFS process.

The space complexity of the code is also `O(m * n)` because it requires a queue to store at least one entire level of the grid's cells, and each cell is stored once. Also, the `ans` matrix of size `m * n` is maintained to store the heights of each cell in the final output.