Quick Navigation

★★★★☆ Average Rating: 4.11 (45 votes) Premium

Solution

Overview

Matrices such as this one are a type of **graph** representation. Standard graph traversal algorithms such as BFS and DFS can be used to solve this problem. If you aren't familiar with these algorithms, check out the Explore Card on BFS.

The naive approach would be to check every cell - that is, iterate through every cell, and at each one, start a traversal that follows the problem's conditions. That is, find every cell that manages to reach both oceans.

This approach, however, is extremely slow, as it repeats a ton of computation. Instead of looking for every path from cell to ocean, let's start at the oceans and try to work our way to the cells. This will be much faster because when we start a traversal at a cell, whatever result we end up with can be applied to only that cell. However, when we start from the ocean and work backwards, we already know that every cell we visit must be connected to the ocean.

Approach 1: Breadth First Search (BFS)

Intuition

If we start traversing from the ocean and flip the condition (check for higher height instead of lower height), then we know that every cell we visit during the traversal can flow into that ocean. Let's start a BFS traversal from every cell that is immediately beside the Pacific ocean, and figure out what cells can flow into the Pacific. Then, let's do the exact same thing with the Atlantic ocean. At the end, the cells that end up connected to both oceans will be our answer.

1	2	2	3	5	1	1	1	3
3	2	3	4	4	2	2	2	3
2	4	5	3	2	1	5	1	4
6	7	1	4	5	1	6	4	2
5	1	1	2	4	4	1	1	4

Initial Board State

1 / 6

Algorithm

1. If the input is empty, immediately return an empty array.
2. Initialize variables that we will use to solve the problem:
   - Number of rows and columns in our matrix;
   - 2 queues, one for the Atlantic Ocean and one for the Pacific Ocean that will be used for BFS;
   - 2 data structures, again one for each ocean, that we'll use to keep track of cells we already visited, to avoid infinite loops;
   - A small array `[(0, 1), (1, 0), (-1, 0), (0, -1)]` that will help with BFS.
3. Figure out all the cells that are adjacent to each ocean, and fill the respective data structures with them.
4. Perform BFS from each ocean. The data structure used to keep track of cells already visited has a double purpose - it also contains every cell that can flow into that ocean.
5. Find the intersection, that is all cells that can flow into both oceans.

Implementation

Putting it all together for the final solution:

Java | Python3 Copy

```java
        for (int i = 0; i < numCols; i++) {
            pacificQueue.offer(new int[]{0, i});
            atlanticQueue.offer(new int[]{numRows - 1, i});
        }

        // Perform a BFS for each ocean to find all cells accessible by each ocean
        boolean[][] pacificReachable = bfs(pacificQueue);
        boolean[][] atlanticReachable = bfs(atlanticQueue);

        // Find all cells that can reach both oceans
        List<List<Integer>> commonCells = new ArrayList<>();
        for (int i = 0; i < numRows; i++) {
            for (int j = 0; j < numCols; j++) {
```

Testcase | Run Code Result | Debugger

Accepted Runtime: 0 ms

Your input
```
[[1,2,2,3,5],[3,2,3,4,4],
[2,4,5,3,1],[6,7,1,4,5],
```

Output
```
[[0,4],
[1,3],[1,4],
```
Diff

Expected
```
[[0,4],[1,3],[1,4],[2,2],
[3,0],[3,1],[4,0]]
```

Console ▴ Use Example Testcases

```java
                if (pacificReachable[i][j] && atlanticReachable[i][j]) {
                    commonCells.add(List.of(i, j));
                }
            }
        }
        return commonCells;
    }

    private boolean[][] bfs(Queue<int[]> queue) {
        boolean[][] reachable = new boolean[numRows][numCols];
        while (!queue.isEmpty()) {
            int[] cell = queue.poll();
            // This cell is reachable, so mark it
            reachable[cell[0]][cell[1]] = true;
            for (int[] dir : DIRECTIONS) { // Check all 4 directions
```

Complexity Analysis

- Time complexity: $O(M \cdot N)$, where M is the number of rows and N is the number of columns.

  In the worst case, such as a matrix where every value is equal, we would visit every cell twice. This is because we perform 2 traversals, and during each traversal, we visit each cell exactly once. There are $M \cdot N$ cells total, which gives us a time complexity of $O(2 \cdot M \cdot N) = O(M \cdot N)$.

- Space complexity: $O(M \cdot N)$, where M is the number of rows and N is the number of columns.

  The extra space we use comes from our queues, and the data structure we use to keep track of what cells have been visited. Similar to the time complexity, for a given ocean, the amount of space we will use scales linearly with the number of cells. For example, in the Java implementation, to keep track of what cells have been visited, we simply used 2 matrices that have the same dimensions as the input matrix.

  The same logic follows for the queues - we can't have more cells in the queue than there are cells in the matrix!

Approach 2: Depth First Search (DFS)

Intuitively, BFS makes more sense for this problem since water flows in the same manner. However, we can also use DFS, and it doesn't really make much of a difference. So, if you prefer DFS, then that's perfectly fine for this problem. Additionally, it's possible that your interviewer will ask you to implement the problem recursively instead of iteratively. Recursion must be DFS, not BFS.

Algorithm

DFS is very similar to BFS. Instead of using a queue and working iteratively, we'll use recursion. Our `dfs` method will be called for every reachable cell. Note: we could also work iteratively with DFS, in which case we would simply use a stack instead of a queue like in the Approach 1 code, with mostly everything else being identical to the BFS approach.

Implementation

Java | Python3 | Copy

```java
class Solution {
    private static final int[][] DIRECTIONS = new int[][]{{0, 1}, {1, 0}, {-1, 0}, {0, -1}};
    private int numRows;
    private int numCols;
    private int[][] landHeights;

    public List<List<Integer>> pacificAtlantic(int[][] matrix) {
        // Check if input is empty
        if (matrix.length == 0 || matrix[0].length == 0) {
            return new ArrayList<>();
        }

        // Save initial values to parameters
        numRows = matrix.length;
        numCols = matrix[0].length;
        landHeights = matrix;
        boolean[][] pacificReachable = new boolean[numRows][numCols];
        boolean[][] atlanticReachable = new boolean[numRows][numCols];

        // Loop through each cell adjacent to the oceans and start a DFS
        for (int i = 0; i < numRows; i++) {
            dfs(i, 0, pacificReachable);
            dfs(i, numCols - 1, atlanticReachable);
        }
        for (int i = 0; i < numCols; i++) {
            dfs(0, i, pacificReachable);
            dfs(numRows - 1, i, atlanticReachable);
```

Complexity Analysis

- Time complexity: $O(M \cdot N)$, where M is the number of rows and N is the number of columns.

  Similar to approach 1. The `dfs` function runs exactly once for each cell accessible from an ocean.

- Space complexity: $O(M \cdot N)$, where M is the number of rows and N is the number of columns.

  Similar to approach 1. Space that was used by our queues is now occupied by `dfs` calls on the recursion stack.

Report Article Issue

Comments: 34 | Best | Most Votes | Newest to Oldest | Oldest to Newest

Type comment here... (Markdown is supported)

Post

codersaif ★ 228 March 12, 2021 6:01 PM

"Water can only flow in four directions"
Here water means, land water, not ocean water.
Hope that will help someone understanding the problem

118 Show 1 reply Reply

mehakwaliaC 269 Last Edit: March 11, 2021 8:04 AM

I legit spent 3 hours on this one! the implementation is tough!

51 Show 5 replies Reply

tusharjaiswal 112 March 25, 2021 1:48 AM

problem statement is ambiguous

112 Show 2 replies Reply

Ram_Babu 147 March 25, 2021 2:02 AM

This problem description is too ambiguous.

64 Reply

goyalashish 30 March 25, 2021 2:33 PM

please refine the problem statement & give some explanation around the test cases.
I solved it but it took me an hour to understand the problem & seeing others comment seems like it's a problem faced by everyone.

21 Reply

vivek404 36 March 25, 2021 10:51 AM

bad problem description but nice editorial!, after reading the overview section of the editorial I understood the problem.

12 Reply

lchelloroad 15 March 25, 2021 10:47 PM

the description is really not clear.
I was stuck on DP first, then after failed on a few cases, I realized it is typical DFS problem.

11 Reply

wesbz 8 April 3, 2021 10:29 AM

This problem is poorly explained. Please work on that.

4 Reply

ryz 295 Last Edit: March 25, 2021 6:45 PM

For anyone that wondering why we dont need a `visited` hashset(or matrix) in addition to `reachable` to mark those points that are visited by a single DFS/BFS path, its because there will be NO circle, either the visited cell are marked true by `reachable` or not true if the value is smaller than previous value (one of termination condition)

3 Reply

SquirrelN 77 Last Edit: March 26, 2021 10:25 AM

Really love the problem, however, like many have said - poor job explaining it though.

2 Reply