2290. Minimum Obstacle Removal to Reach Corner

Hard | Breadth-First Search | Graph | Array | Matrix | Shortest Path | Heap (Priority Queue) | Leetcode Link

Problem Description

You are given a 2D integer array `grid` with two possible values in each cell:

- `0` represents an empty cell you can move through.
- `1` represents an obstacle that can be removed.

The grid has `m` rows and `n` columns, and the goal is to move from the top-left corner `(0, 0)` to the bottom-right corner `(m - 1, n - 1)`. You can move in four directions from an empty cell: up, down, left, or right.

The problem asks you to find the minimum number of obstacles (`1`s) that need to be removed to enable this path-finding from the start to the end.

Intuition

To solve this problem, we use a breadth-first search (BFS) approach. BFS is a graph traversal method that expands and examines all nodes of a level before moving to the next level. This characteristic makes BFS suitable for finding the shortest path on unweighted graphs, or in this case, to find the minimum number of obstacles to remove.

The idea is to traverse the grid using BFS, keeping track of the position `(i, j)` we're currently in and the count `k` of obstacles that we have removed to reach this position. Each time we can either move to an adjacent empty cell without incrementing the obstacle removal count or move to an adjacent cell with an obstacle by incrementing the removal count by one.

We use a deque to facilitate the BFS process. When we encounter an empty cell, we add the position to the front of the deque, giving priority to such moves for expansion before those requiring obstacle removal, effectively ensuring the BFS prioritizes paths with fewer obstacles removed.

A visited set `vis` is used to keep track of the cells we have already processed to avoid re-processing and potentially getting into cycles.

The search continues until we reach the bottom-right corner `(m - 1, n - 1)`, at which point we return the count of removed obstacles `k` which represents the minimum number of obstacles that need to be removed.

Solution Approach

The provided Python code implements the BFS algorithm using a deque, a double-ended queue that allows the insertion and removal of elements from both the front and the back with constant time complexity `O(1)`.

BFS (Breadth-First Search)

At a high level, the BFS algorithm works by visiting all neighbors of a node before moving on to the neighbors' neighbors. It uses a queue to keep track of which nodes to visit next.

Deque for BFS

In the Python code, the `deque` is used instead of a regular queue. When the BFS explores a cell with an obstacle (value `1`), it appends the new state to the end of the deque. Conversely, when it explores a cell without an obstacle (value `0`), it appends the new state to the front of the deque.

Implementation Details

1. Initialize the deque `q` with a tuple containing the starting position and the initial number of obstacles removed `(0, 0, 0)`.
2. Create a `vis` set to record visited positions and avoid revisiting them.
3. The `dirs` tuple is used to calculate the adjacent cells' positions with the help of a helper function like `pairwise`.
4. Enter a loop that continues until the end condition is met (reaching `(m - 1, n - 1)`).
5. Use `popleft` to get the current position and the count `k` of the obstacles removed.
6. If the target position is reached, return `k`.
7. If the position has been visited before (`(i, j) in vis`), just continue to the next iteration.
8. Otherwise, mark the position as visited by adding `(i, j)` to `vis`.
9. Iterate over all possible directions, and calculate the adjacent cell positions `(x, y)`.
10. If `(x, y)` is within the grid bounds and is not an obstacle, append it to the front of the deque with the same number of removed obstacles `k`.
11. If `(x, y)` has an obstacle, append it to the end of the deque and increment the count of removed obstacles by `1`.

This process prioritizes exploring paths with fewer obstacles, and since BFS is used, it guarantees that the first time we reach the bottom-right corner is via the path that requires the minimum number of obstacles to be removed.

The solution effectively mixes BFS traversal with a priority queue concept by using a deque to manage traversal order, ensuring an efficient search for the minimum obstacle removal path.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we have the following `grid`:

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

Here, `m = 3` and `n = 3`. We want to move from `(0, 0)` to `(2, 2)` with the minimum number of obstacles removed.

Step-by-Step Process:

1. We start by initializing the deque `q` with the starting position and the initial number of obstacles removed, which is `0`. So `q = deque([(0, 0, 0)])`.
2. The `vis` set is initialized to ensure we don't visit the same positions repeatedly. In the beginning, it's empty: `vis = set()`.
3. The BFS begins. We dequeue the first element `(i, j, k) = (0, 0, 0)`, where `(i, j)` represents the current cell, and `k` is the number of removed obstacles so far.
4. We have not reached the target, so we check the adjacent cells:
   - Right `(0, 1)`: It has an obstacle. We increment obstacle count and add it to the end of `q`: `q = deque([(0, 1, 1)])`.
   - Down `(1, 0)`: No obstacle. We add it to the front of `q`: `q = deque([(1, 0, 0), (0, 1, 1)])`.
5. We mark the current cell `(0, 0)` as visited: `vis.add((0, 0))`.
6. Now the deque `q` has cells to process, so we take the one from the front, which is `(1, 0, 0)`.
7. From `(1, 0)`, we again check the adjacent cells:
   - Right `(1, 1)`: It has an obstacle, add it to end: `q = deque([(0, 1, 1), (1, 1, 1)])`.
   - Down `(2, 0)`: No obstacle, add it to front: `q = deque([(2, 0, 0), (0, 1, 1), (1, 1, 1)])`.
8. We mark the cell `(1, 0)` as visited: `vis.add((1, 0))`.
9. The steps repeat, dequeuing from the front, checking adjacent cells, and enqueueing in the deque according to the rules.
10. Eventually, we dequeue the cell `(2, 0, 0)`. From here, we can go right to `(2, 1)` with no obstacle and add it to the front: `q = deque([(2, 1, 0), (0, 1, 1), (1, 1, 1)])`.
11. Again, we mark `(2, 0)` as visited: `vis.add((2, 0))`.
12. Dequeuing `(2, 1, 0)`, we can go right to the target `(2, 2)` with no obstacle. We add it to the front.
13. Mark `(2, 1)` as visited and check the next cell from `q`.
14. Finally, we reach `(2, 2)` which is the target. We have not needed to remove any obstacles, so we return `k = 0`.

Throughout this process, we have explored paths with the fewest obstacles first, using the deque to effectively prioritize cells without obstacles. By doing this, we've ensured that the first time we reach the end, it is the path with the minimum number of obstacles removed. In this case, no obstacles needed to be removed.

Python Solution

```python
from collections import deque
from itertools import pairwise  # Note: requires Python 3.10+, for earlier versions use a custom pairwise implementation

class Solution:
    def minimumObstacles(self, grid):
        # Get the dimensions of the grid
        rows, cols = len(grid), len(grid[0])

        # Initialize a queue with the starting point and 0 obstacles encountered
        queue = deque([(0, 0, 0)])

        # Create a set to keep track of visited cells
        visited = set()

        # Define the directions to move in the grid, pairwise will use this
        directions = (-1, 0, 1, 0, -1)

        # Loop until we find the exit or run out of cells to explore
        while queue:
            # Pop the cell from the queue and count of obstacles encountered so far
            i, j, obstacle_count = queue.popleft()

            # If we've reached the bottom right corner, return the obstacle count
            if i == rows - 1 and j == cols - 1:
                return obstacle_count

            # If this cell has been visited before, skip to the next iteration
            if (i, j) in visited:
                continue

            # Mark the current cell as visited
            visited.add((i, j))

            # Iterate over all possible moves (up, down, left, right)
            for dx, dy in pairwise(directions):
                x, y = i + dx, j + dy

                # Check if the new position is within bounds
                if 0 <= x < rows and 0 <= y < cols:
                    # If there is no obstacle, add the cell to the front of the queue to explore it next
                    if grid[x][y] == 0:
                        queue.appendleft((x, y, obstacle_count))
                    # If there is an obstacle, add the cell to the back of the queue with the obstacle count incremented
                    else:
                        queue.append((x, y, obstacle_count + 1))

# If running Python version earlier than 3.10, define your pairwise function like this:
# def pairwise(iterable):
#     "s -> (s0,s1), (s1,s2), (s2, s3), ..."
#     a, b = tee(iterable)
#     next(b, None)
#     return zip(a, b)

# Note: The pairwise function returns consecutive pairs of elements from the input.
# For the directions in this code, it's used to generate the pairs (up, right), (right, down),
# (down, left), and (left, up) to traverse the grid in a clockwise manner.
```

Java Solution

```java
class Solution {
    public int minimumObstacles(int[][] grid) {
        // Get the dimensions of the grid
        int rows = grid.length, cols = grid[0].length;

        // Create a deque to hold the positions and the current obstacle count
        Deque<int[]> queue = new ArrayDeque<>();
        // Start from the upper left corner (0,0) with 0 obstacles
        queue.offer(new int[] {0, 0, 0});
        // Array to iterate over the 4 possible directions (up, right, down, left)
        int[] directions = {-1, 0, 1, 0, -1};
        // Visited array to keep track of positions already visited
        boolean[][] visited = new boolean[rows][cols];

        // Process cells until the queue is empty
        while (!queue.isEmpty()) {
            // Poll the current position and the number of obstacles encountered so far
            int[] position = queue.poll();
            int currentRow = position[0];
            int currentCol = position[1];
            int obstacles = position[2];

            // Check if we have reached the bottom-right corner
            if (currentRow == rows - 1 && currentCol == cols - 1) {
                // If we reached the destination, return the number of obstacles encountered
                return obstacles;
            }

            // If we have already visited this cell, skip it
            if (visited[currentRow][currentCol]) {
                continue;
            }

            // Mark the current cell as visited
            visited[currentRow][currentCol] = true;

            // Explore the neighboring cells
            for (int h = 0; h < 4; ++h) {
                int nextRow = currentRow + directions[h];
                int nextCol = currentCol + directions[h + 1];

                // Check the boundaries of the grid
                if (nextRow >= 0 && nextRow < rows && nextCol >= 0 && nextCol < cols) {
                    // If the next cell is free (no obstacle)
                    if (grid[nextRow][nextCol] == 0) {
                        // Add it to the front of the queue to be processed with the same obstacle count
                        queue.offerFirst(new int[] {nextRow, nextCol, obstacles});
                    } else {
                        // If there's an obstacle, add it to the end of the queue with the obstacle count incremented by 1
                        queue.offerLast(new int[] {nextRow, nextCol, obstacles + 1});
                    }
                }
            }
        }
        // We include a return statement to satisfy the compiler, although the true return occurs inside the loop
        return -1; // This will never be reached as the problem guarantees a path exists
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <deque>
#include <tuple>
#include <cstring>

class Solution {
public:
    // This function computes the minimum number of obstacles that need to be removed
    // to move from the top-left corner to the bottom-right corner of the grid.
    int minimumObstacles(std::vector<std::vector<int>>& grid) {
        int rows = grid.size();          // The number of rows in the grid
        int cols = grid[0].size();       // The number of columns in the grid

        // deque to perform BFS with a slight modification to handle 0-cost and 1-cost paths
        std::deque<std::tuple<int, int, int>> queue;
        queue.emplace_back(0, 0, 0);     // Start from the top-left corner (0,0)

        // Visited array to keep track of visited cells
        bool visited[rows][cols];
        std::memset(visited, 0, sizeof(visited));

        // Direction vectors for up, right, down and left movements
        int directions[5] = {-1, 0, 1, 0, -1};

        while (!queue.empty()) {
            // Get the current cell's row, column, and obstacle count
            auto [currentRow, currentCol, obstacleCount] = queue.front();
            queue.pop_front();

            // If we have reached the bottom-right corner, return the obstacle count
            if (currentRow == rows - 1 && currentCol == cols - 1) {
                return obstacleCount;
            }

            // Skip this cell if it's already been visited
            if (visited[currentRow][currentCol]) {
                continue;
            }

            // Mark the current cell as visited
            visited[currentRow][currentCol] = true;

            // Loop through the possible movements using the direction vectors
            for (int dir = 0; dir < 4; ++dir) {
                int newRow = currentRow + directions[dir];
                int newCol = currentCol + directions[dir + 1];

                // Check if the new position is valid and within the grid boundaries
                if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols) {
                    // If new cell has no obstacles, it has the same obstacle count 'k'
                    // and it is pushed to the front to give it higher priority
                    if (grid[newRow][newCol] == 0) {
                        queue.emplace_front(newRow, newCol, obstacleCount);
                    }
                    // If new cell has an obstacle, increase the obstacle count 'k' by 1
                    // and it is pushed to the back
                    else {
                        queue.emplace_back(newRow, newCol, obstacleCount + 1);
                    }
                }
            }
        }

        // If the function somehow reaches here (it should not), return -1 as an error signal.
        // This return statement is logically unreachable because the loop is guaranteed to break
        // when the bottom-right corner is reached.
        return -1;
    }
};
```

Typescript Solution

```typescript
function minimumObstacles(grid: number[][]): number {
    const rows = grid.length,
        columns = grid[0].length; // Define the number of rows and columns of the grid
    const directions = [ // Directions for traversal: right, left, down, up
        [0, 1],
        [0, -1],
        [1, 0],
        [-1, 0],
    ];
    let obstacleCount = Array.from({ length: rows }, () => new Array(columns).fill(Infinity)); // Initialize obstacle count for eac
    obstacleCount[0][0] = 0; // Starting point has 0 obstacles
    let deque = [[0, 0]]; // Double-ended queue to keep track of which cell to visit next

    while (deque.length > 0) {
        const [currentX, currentY] = deque.shift(); // Extract current cell coordinates

        for (const [dx, dy] of directions) {
            const [nextX, nextY] = [currentX + dx, currentY + dy]; // Calculate adjacent cell coordinates

            if (nextX < 0 || nextX >= rows || nextY < 0 || nextY >= columns) {
                continue; // Out of bounds check
            }

            const currentCost = grid[nextX][nextY]; // Cost of the adjacent cell (0 for open cell, 1 for cell with an obstacle)

            if (obstacleCount[currentX][currentY] + currentCost >= obstacleCount[nextX][nextY]) {
                continue; // If the new path isn't better, continue to the next adjacent cell
            }

            obstacleCount[nextX][nextY] = obstacleCount[currentX][currentY] + currentCost; // Update the obstacle count for the cel

            if (currentCost === 0) {
                deque.unshift([nextX, nextY]); // If no obstacle, add to the front of the queue
            } else {
                deque.push([nextX, nextY]); // If there's an obstacle, add to the back of the queue
            }
        }
    }

    return obstacleCount[rows - 1][columns - 1]; // Returns the minimum number of obstacles to reach the bottom-right corner
}
```

Time and Space Complexity

Time Complexity

The time complexity of the algorithm can be estimated by considering the number of operations it performs. The algorithm uses a queue that can have at most every cell `(i, j)` from the grid enqueued, and the grid has `m * n` cells. Each cell is inserted into the queue at most once, since we're using a set of visited nodes to prevent re-visitation.

The `deque` operations `appendleft` and `popleft` have `O(1)` time complexity. The for loop executes at most four times for each cell, corresponding to the four possible directions one can move in the grid.

Considering all of this, the overall time complexity is `O(m * n)`, as every cell is considered at most once.

Space Complexity

The space complexity consists of the space needed for the queue and the set of visited cells.

The queue can store up to `m * n` elements, if all cells are inserted, and the set will at the same time contain a maximum of `m * n` elements as well to track visited cells. The sum is `2 * m * n` but in terms of Big O notation, the constants are ignored.

Thus, the space complexity is `O(m * n)` for the queue and the visited set combined.