Minimum Obstacle Removal To Reach Corner

solved by	Senan
Platform	LeetCode
difficulty	Hard
Serial	2290
tags	BFS Dijkstras Algorithm Dynamic Programming
language	C++
solved on	@28/11/2024
link	https://leetcode.com/problems/minimum-obstacle-removal-to-reach-corner/description/
Completion	☑

Intuition

The goal is to find the minimum number of obstacles that need to be removed to travel from the top-left to the bottom-right of the grid. The problem can be approached as a shortest path problem where each cell is either an obstacle (weight 1) or free space (weight 0). The solution uses Dijkstra's algorithm to minimize the "cost" of reaching each cell.

Approach

1. **Graph Representation**:
    - Treat each cell as a node and edges connecting adjacent cells have weights determined by whether they are obstacles or not.
    - A `map` is used to store the graph with each cell's neighbors and their weights.
2. **Priority Queue**:
    - Use a min-heap (`priority_queue`) to always explore the least costly path first, prioritizing cells with fewer obstacle removals.
3. **Dijkstra's Algorithm**:
    - Start from `(0, 0)` with an initial cost of `0`.
    - Update the cost to reach neighboring cells if a shorter path is found.
    - Continue until reaching the bottom-right cell `(n-1, m-1)`.
4. **Distance Array**:
    - Use `dist` to track the minimum cost to reach each cell. Initialize with a high value (`1e9`).

Complexity

Time Complexity:

- **O(n * m * log(n * m))**: Each cell is processed once, and each insertion into the priority queue is logarithmic in the total number of cells.

Space Complexity:

- **O(n * m)**: For the graph representation and the distance array.

Code

```
class Solution {
public:
    int minimumObstacles(vector<vector<int>>& grid) {
        int n = grid.size(), m = grid[0].size();
        map<pair<int, int>, vector<pair<pair<int, int>, int>>> graph;

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (j < m - 1) graph[{i, j}].push_back({{i, j + 1}, grid[i][j + 1]});
                if (i < n - 1) graph[{i, j}].push_back({{i + 1, j}, grid[i + 1][j]});
                if (j > 0) graph[{i, j}].push_back({{i, j - 1}, grid[i][j - 1]});
                if (i > 0) graph[{i, j}].push_back({{i - 1, j}, grid[i - 1][j]});
            }
        }

        priority_queue<pair<int, pair<int, int>>,
                                         vector<pair<int, pair<int, int>>>,
                                         greater<>> q;

        q.push({0, {0, 0}});
        vector<vector<int>> dist(n, vector<int>(m, 1e9));
        dist[0][0] = 0;

        while (!q.empty()) {
            auto [distance, node] = q.top();
            q.pop();
            int i = node.first, j = node.second;

            for (auto &[adj, weight] : graph[{i, j}]) {
                int x = adj.first, y = adj.second;
                if (distance + weight < dist[x][y]) {
                    dist[x][y] = distance + weight;
                    q.push({dist[x][y], {x, y}});
                }
            }
        }

        return dist[n - 1][m - 1];
    }
};
```