2061. Number of Spaces Cleaning Robot Cleaned

Medium Array Matrix Simulation Leetcode Link

Problem Description

The given problem is about simulating the path of a cleaning robot inside a room. The room is represented as a 0-indexed 2D binary matrix `room` where a 0 indicates an empty space that can be cleaned, and a 1 indicates a space with an object that cannot be passed through. The robot starts at the top left corner of the matrix, which is guaranteed to be an empty space (a 0), and initially moves right.

The behavior of the robot is very specific: it proceeds straight until it either hits an object (a 1 in the matrix) or it reaches the edge of the room. Upon encountering an obstruction or the edge, the robot turns 90 degrees clockwise and continues in the new direction. Importantly, any empty space the robot passes over is considered "cleaned," including the starting space.

The problem asks for the total number of unique spaces that the robot cleans if it moves indefinitely following these rules.

Intuition

The key to solving this problem is to simulate the movement of the robot and keep track of the spaces it cleans. We need a way to mark the spaces the robot visits as cleaned, prevent re-cleaning the same space, and we want to ensure the robot turns appropriately when required.

To solve this problem, we can perform a Depth-First Search (DFS) algorithm starting from the initial position of the robot. The DFS function can be implemented to take the current position (i, j) and the direction `k` the robot is facing as its parameters. We will clean the space if it's empty (represented by a 0), mark it as visited (to avoid re-cleaning), and then proceed in the current direction if the next space is also empty (a 0). If the robot encounters an object (a 1) or the edge of the room, it should turn 90 degrees clockwise, which can be implemented by updating the direction `k`.

For our DFS function, we keep track of "visited" states including positions and the current direction. This is important because the robot could potentially return to a position it has visited before but facing a different direction. We need to ensure we only count a space as clean when we visit it for the first time in the given direction.

We also keep a count of the cleaned spaces, incrementing it whenever we clean a new space. The DFS proceeds recursively, and we return the final count of cleaned spaces once the simulation is complete and there are no new spaces left to visit.

This approach works as the robot's path is deterministic, and by following the robot's rules, we can simulate its entire path until it either gets stuck in a loop or has cleaned all accessible spaces.

Solution Approach

The implementation of the solution uses the Depth-First Search (DFS) algorithm as its base approach to traverse through the room matrix. The algorithm deploys a recursive function `dfs` which takes the current position `(i, j)` and the current facing direction `k` as its parameters.

The algorithm uses several variables and a structure:

- `dir` is a tuple `(0, 1, 0, -1, 0)` representing the directions in which the robot can move. Index `k` indicates the current direction. If `k` is 0, it means the robot moves to the right; if `k` is 1, it means moving down; if `k` is 2, moving left; and if `k` is 3, moving up.
- `vis` is a set that keeps track of the visited positions and their associated directions. A position is deemed visited along with a specific direction, not just the positions itself.
- `ans` is the accumulator counter that counts the number of unique spaces that have been cleaned.

Let's walk through the implementation details of the solution:

1. We start off with our DFS function `dfs(i, j, k)`.
   - If the position `(i, j, k)` is in `vis`, it means this space with the current direction was already processed, and we return immediately to avoid repeating work.
   - If the current space is clean (`room[i][j] == 0`), we increment `ans` to denote a new clean space, and we mark the space as visited by setting `room[i][j]` to `-1`.
   - The current state `(i, j, k)` is added to the `vis` set to mark it as visited.
   - The robot then tries to move in the direction `k`. If the next position is within the bounds of the matrix and is empty, the robot moves to it, and `dfs` is called recursively.
   - If the robot encounters an obstruction or the edge, it turns 90 degrees clockwise by incrementing `k (k + 1) % 4`, and `dfs` is called recursively without changing the position but only changing the direction.
2. The initial call to `dfs(0, 0, 0)` begins the simulation with the robot starting in the upper left corner and facing right.
3. The `dfs` function will continue to recurse, simulating the robot's movement until all cleanable spaces are visited.
4. Eventually, the program will run out of new spaces to visit. At this point, all spaces that could be cleaned by the robot following the rules have been counted, and the final `ans` value denotes this count.

This recursive algorithm with the direction and visited tracking elegantly simulates the robot's movement and provides us with the correct count of uniquely cleaned spaces. The solution works efficiently within the constraints specified in the problem.

Example Walkthrough

Let's consider a small room `room` with the following scenario to illustrate the solution approach:

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

In this room matrix, `0` denotes an empty and cleanable space, while `1` represents an obstacle the robot can't pass through.

We will simulate the robot's movement starting from the top-left corner `(0, 0)`, initially moving right (direction `k` = 0).

1. The robot starts at `(0, 0)` which is empty, and we perform `dfs(0, 0, 0)`. The space is counted as cleaned (`ans = 1`), and the position `(0, 0, 0)` is marked as visited.
2. The robot moves right to `(0, 1)` following its rightward direction. Another `dfs(0, 1, 0)` is called. The empty space is cleaned (`ans = 2`), and `(0, 1, 0)` is marked as visited.
3. The robot tries to move right again but encounters a wall `(room[0][2] == 1)`. It can't proceed, so it turns 90 degrees clockwise and faces down (`k = 1`).
4. Now facing down, the robot moves to `(1, 1)` since the space below the starting point `(0, 0)` is an obstacle. We perform `dfs(1, 1, 1)`. `(1, 1, 1)` is marked as visited, and since we encountered another obstacle at `(1, 1)`, the robot turns 90 degrees clockwise again (`k = 2`).
5. Facing left (`k = 2`), the robot moves to `(1, 0)` and we call `dfs(1, 0, 2)`. The space is cleanable, so `ans` is updated (`ans = 3`), and `(1, 0, 2)` is marked as visited.
6. Since `(0, 0)` has been visited from the right, the robot will move left another space to `(2, 0)`. This clean space updates `ans` (`ans = 4`), and `(2, 0, 2)` is marked as visited.
7. Continuing left would hit the matrix bound, so the robot turns clockwise, now facing up (`k = 3`), but this is blocked by an obstacle. Clockwise again (`k = 0`), this next space is `(2, 1)`, but it's blocked too, so it turns clockwise (`k = 1`).
8. Now facing down, the robot can move into `(2, 1)` and perform `dfs(2, 1, 1)`, cleaning this space (`ans = 5`), and marking `(2, 1, 1)` as visited.
9. Next, it advances to `(2, 2)` and calls `dfs(2, 2, 1)`, cleans it (`ans = 6`), and marks `(2, 2, 1)` as visited. This is the last horizontal line where the robot can move without obstacles, so it continues right until it hits the boundary, turns clockwise to face up (`k = 3`), and moves back.
10. On moving up, at `(1, 2)`, it can't move right, left, or up since either the space is an obstacle or has been visited from that direction already. The robot is stuck in a loop.

The system comes to a halt when there are no new spaces to be visited. The answer, `ans`, is `6`, which represents the total number of unique cleanable spaces in the room.

Python Solution

```python
class Solution:
    def numberOfCleanRooms(self, room):
        # Inner recursive function to perform depth-first search
        def dfs(row, col, direction):
            # If this state has been visited before, do nothing
            if (row, col, direction) in visited:
                return

            # If the current room is clean, increment the counter
            nonlocal clean_rooms_count
            clean_rooms_count += room[row][col] == 0

            # Mark this room as visited by setting its value to -1
            room[row][col] = -1

            # Add the current state to the visited set
            visited.add((row, col, direction))

            # Calculate the next room's coordinates based on the direction
            next_row, next_col = row + directions[direction], col + directions[direction + 1]

            # If the next room is within bounds and not a wall (room value != 1), move to it
            if 0 <= next_row < len(room) and 0 <= next_col < len(room[0]) and room[next_row][next_col] != 1:
                dfs(next_row, next_col, direction)
            else:
                # If there's a wall, change direction (rotate right)
                dfs(row, col, (direction + 1) % 4)

        # Set to keep track of visited states (row, col, direction)
        visited = set()

        # Directions correspond to right (0, 1), down (1, 0), left (0, -1), up (-1, 0)
        # The directions are intentionally shifted by one to facilitate indexing. This means that
        # directions[i] gives the row offset and directions[i+1] gives the column offset.
        directions = [0, 1, 0, -1, 0]

        # Counter for the number of clean rooms visited
        clean_rooms_count = 0

        # Start the DFS from the top-left corner of the room facing right (direction is 0)
        dfs(0, 0, 0)

        # Return the total count of clean rooms visited
        return clean_rooms_count
```

Java Solution

```java
class Solution {
    private boolean[][][] visited;
    private int[][] room;
    private int cleanRoomsCount;

    // Counts the number of clean rooms that the robot cleans
    public int numberOfCleanRooms(int[][] room) {
        // The visited array tracks the cells visited for each of the 4 directions
        visited = new boolean[room.length][room[0].length][4];
        this.room = room; // Initialize room reference
        cleanRoomsCount = 0; // Initialize clean rooms count

        dfs(0, 0, 0); // Start the DFS from the top-left corner in direction '0' (right)
        return cleanRoomsCount; // Return the total number of clean rooms visited
    }

    // Performs DFS to navigate the room based on the given rules
    private void dfs(int i, int j, int dir) {
        // If current position and direction is already visited, do nothing
        if (visited[i][j][dir]) {
            return;
        }

        // Relative directions: right (0), down (1), left (2), up (3)
        int[] directions = {0, 1, 0, -1, 0};

        // Mark the current position and direction as visited
        visited[i][j][dir] = true;

        // If current room is clean (i.e., not a wall or already counted), increment cleanRoomsCount
        if (room[i][j] == 0) {
            cleanRoomsCount++;
            // Mark the room as cleaned by setting it to a distinct value (-1)
            // to avoid recounting when visited from a different direction
            room[i][j] = -1;
        }

        // Compute next position based on the current direction
        int nextX = i + directions[dir];
        int nextY = j + directions[dir + 1];

        // Move to next position if it's within bounds and is not a wall
        if (nextX >= 0 && nextX < room.length && nextY >= 0 && nextY < room[0].length && room[nextX][nextY] != 1) {
            dfs(nextX, nextY, dir); // Continue in the same direction
        } else {
            // If hitting a wall or out of bounds, turn right (increment direction)
            // and continue DFS with the new direction. Use modulo to wrap direction.
            dfs(i, j, (dir + 1) % 4);
        }
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <functional>
#include <cstring>

class Solution {
public:
    int numberOfCleanRooms(vector<vector<int>>& room) {
        int m = room.size();    // Number of rows
        int n = room[0].size(); // Number of columns
        bool visited[m][n][4];  // 3D array to keep track of visited states
        memset(visited, false, sizeof(visited)); // Initialize visited array to false

        // Array that represents the directions we can move in the grid (right, down, left, up)
        int directions[5] = {0, 1, 0, -1, 0};

        int cleanRoomsCount = 0; // Initialize count of clean rooms

        // Lambda function that performs Depth-First Search (DFS) starting from (i, j) in direction k
        function<void(int, int, int)> dfs = [&](int i, int j, int dirIndex) {
            // If the current state has been visited, return
            if (visited[i][j][dirIndex]) {
                return;
            }

            // Mark the current cell as visited and clean it if it's not an obstacle
            cleanRoomsCount += room[i][j] == 0;
            room[i][j] = -1; // Mark the room as cleaned, indicating that it's been traversed
            visited[i][j][dirIndex] = true; // Mark current state as visited

            // Calculate the next cell coordinates by moving in the current direction
            int nextX = i + directions[dirIndex];
            int nextY = j + directions[dirIndex + 1];

            // If the next cell is within bounds and not an obstacle, move to the next cell
            if (nextX >= 0 && nextX < m && nextY >= 0 && nextY < n && room[nextX][nextY] != 1) {
                dfs(nextX, nextY, dirIndex);
            } else {
                // If we can't move forward, turn right (clockwise) and stay in the current cell
                dfs(i, j, (dirIndex + 1) % 4); // % 4 to cycle through direction indices
            }
        };

        dfs(0, 0, 0); // Start DFS from the top-left corner in the right direction
        return cleanRoomsCount; // Return the total count of clean rooms
    }
};
```

Typescript Solution

```typescript
const DIRECTIONS: number[] = [0, 1, 0, -1, 0]; // Directions: right, down, left, up.

let visited: boolean[][][]; // 3D array to keep track of visited states.
let cleanRoomsCount: number = 0; // Count of clean rooms.

function numberOfCleanRooms(room: number[][]): number {
    const rows: number = room.length; // Number of rows in the room grid.
    const cols: number = room[0].length; // Number of columns in the room grid.
    visited = Array.from({length: rows}, () => Array.from({length: cols}, () => Array(4).fill(false)));

    // Kick off the depth-first search (DFS) from the top-left corner of the room facing right.
    dfs(0, 0, 0, room);
    return cleanRoomsCount; // Return final count of cleaned rooms.
}

function dfs(row: number, col: number, dirIndex: number, room: number[][]): void {
    // If the current state has already been visited, no need to proceed further.
    if (visited[row][col][dirIndex]) {
        return;
    }

    // If the current cell is not an obstacle, increment the clean rooms count.
    if (room[row][col] === 0) {
        cleanRoomsCount++;
    }
    // Mark the current cell visited and cleaned by indicating it's been traversed.
    visited[row][col][dirIndex] = true;
    room[row][col] = -1; // Indication of cleaned room.

    // Calculate next cell's coordinates based on current direction.
    const nextRow = row + DIRECTIONS[dirIndex];
    const nextCol = col + DIRECTIONS[dirIndex + 1];

    // Check if the next cell is within bounds and is not an obstacle.
    if (nextRow >= 0 && nextRow < rows && nextCol >= 0 && nextCol < cols && room[nextRow][nextCol] !== 1) {
        dfs(nextRow, nextCol, dirIndex, room); // Move to the next cell in the same direction.
    } else {
        // If facing an obstacle or out of bounds, turn right (clockwise) without changing cell.
        dfs(row, col, (dirIndex + 1) % 4, room); // % 4 cycles through the direction indices.
    }
}

// Example usage:
// console.log(numberOfCleanRooms([[0,0,0],[0,1,0],[0,0,0]])); // Possible room layout
```

Time and Space Complexity

The time complexity of the provided code is `O(N * M)` where `N` represents the number of rows and `M` represents the number of columns in the `room` grid. This is because in the worst-case scenario, the robot will visit each cell at most once in each direction before it finds itself in a previously visited state, leading to a visitation of all `N * M` cells in 4 directions.

The space complexity is also `O(N * M)` because we are using a set `vis` to store a tuple containing the coordinates and direction for each visited cell. In the worst case, each cell will be visited in all 4 directions, leading to `4 * N * M` possible states that can be stored in the set. However, constants are dropped in Big O notation, leaving us with `O(N * M)`.