489. Robot Room Cleaner

Hard Backtracking Interactive

Leetcode Link

Problem Description

This problem presents a scenario where you are in control of a robot placed in an unknown position within a room. The room is set up as a grid with "1" representing an empty space that the robot can move through, and "0" representing a wall that the robot cannot pass. Your task is to create an algorithm that instructs the robot to clean the entire room, covering every "1" cell in the grid.

The robot is equipped with a set of APIs that allow it to:

1. Move forward into an adjacent cell (`move` method), unless there is a wall which will cause the robot to stay put.
2. Turn left or right by 90 degrees without moving from its current cell (`turnLeft` and `turnRight` methods).
3. Clean the cell it is currently on (`clean` method).

One crucial detail to note is that the initial orientation of the robot is facing "up," and all four edges of the grid are walled off. Furthermore, you must write this algorithm without any knowledge of the room's layout or the robot's starting position.

Intuition

The solution to this problem involves a depth-first search (DFS) strategy. The idea behind DFS is to exhaustively explore a path until hitting an obstacle (in this case, a wall), backtrack, and then try another path. This principle is well-suited for the problem since we need to ensure every navigable cell is visited and cleaned.

Since the robot's starting point is unknown, and the room's layout is also unknown, we can imagine the robot's initial position as the origin of a coordinate system, `(0, 0)`.

For the DFS to work, we need to:

- Keep track of cells that have already been visited to avoid redundant cleaning and to prevent the robot from going in cycles.
- Identify a way to represent the robot's orientation since it can face four different directions (up, right, down, and left). We use a direction vector (`dirs`) for this purpose.

The `dfs` function in the solution recursively explores the grid by trying to move in the current facing direction, cleaning the cell, and then attempting to move forward in the next direction. If a move is successful (indicating an empty cell), the robot moves forward, and the search continues from the new cell position. After exploring one direction, the robot turns 90 degrees to the right to check the next direction.

Once the robot hits a wall and cannot move forward:

- It backtracks to the previous cell by performing a 180-degree turn (`turnRight` twice), moving forward one cell, and then realigning to the original orientation by another 180-degree turn.
- It then continues the DFS from the previous cell in the next direction.

This process continues until all reachable cells have been visited and cleaned.

Solution Approach

The solution uses depth-first search (DFS) to navigate and clean the room. DFS is a popular algorithm for exploring all nodes in a graph or all vertices in a grid by moving as far as possible until you can no longer proceed, then backtracking.

Here's how the algorithm in the provided solution works:

1. **Data Structure for Visited Cells**: A set named `vis` is used to keep track of visited coordinates. This prevents the robot from cleaning the same cell multiple times and ensures that the algorithm terminates.
2. **Directional Handling**: The `dirs` tuple provides the relative coordinates when moving up, right, down, and left respectively, assuming an orientation of facing up initially. These directions correspond to the changes in the robot's coordinates given its current direction.
3. **Recursive DFS Function**: The heart of the solution is the `dfs(i, j, d)` function, which takes the robot's current `x` and `y` coordinates (`i` and `j`), and its current direction (`d`).
   - It starts by cleaning the current cell and adding the cell to the `vis` (visited set).
   - The robot then attempts to move in its current direction and three more directions by turning right sequentially.
   - After attempting to move in each direction:
     - If the move is successful, the robot has discovered a new cell, and `dfs` is called recursively from that new position.
     - After the recursive call (which represents backtracking to the source cell), the robot performs two 90-degree turns to face the opposite direction (`robot.turnRight()` called twice), moves forward to return to the original cell, and then realigns to the initial direction by turning another 180-degrees in total.
   - It then turns right to change to the next direction and continue the DFS.
4. **Triggering DFS**: The DFS begins by calling `dfs(0, 0, 0)` from the robot's initial position, which is treated as the origin `(0, 0)` and its direction as `0`.

Throughout this process, the robot continues to move, turn, and clean until it has reached and cleaned all accessible cells. The `vis` set ensures that the robot never revisits a cell, efficiently covering the entire grid.

By using these approaches, the robot is able to clean the entire room—represented by the `1` cells in the grid—without any prior knowledge of the room's layout or its own initial position.

Example Walkthrough

Imagine a room represented by the following 3×3 grid and the robot starts at the cell marked `R` (which we consider `(0, 0)` for our coordinate system), facing upwards. `1` indicates an empty space, and `0` indicates a wall:

```
1. Room:
2. 1 1 1
3. 1 R 1
4. 0 1 1
```

Step-by-step using the DFS approach:

1. **Initialization**: The robot starts at `(0, 0)`. The cell is automatically cleaned and is added to the visited set `vis`.
2. **Exploration**:
   - The robot tries to move forward (up) into `(0, -1)`. Since it's free, the cell is cleaned, and this position is added to `vis`. The function `dfs(0, -1, 0)` is called.
     From `(0,-1)`, the robot continues and moves up to `(0,-2)`, which is also free. The robot cleans it and then attempts to go right, but there is a wall, so it turns right and tries to go down but hits another wall. It turns right again, and now it moves left, cleans `(0,-1)`, and returns to `(0,-2)` and faces down (original direction).
   - The sequence repeats until it faces upward again, and since all directions are visited, it returns to `(0,-1)`. The robot tries to go right, and it's blocked, so it tries to go down and arrives at `(0,0)`, which is already cleaned and in `vis`.
   - Now the robot tries to move right from `(0,0)`. It can't because there is a wall, so it rotates right and moves down to `(1,0)`. The robot cleans the cell, then tries to go left (hit wall), up (cleaned and in `vis`), right, and finally down. Since down is the only direction it hasn't come from, it moves to `(2,0)` and cleans it.
   - At `(2,0)`, the robot repeats the process: trying each direction, cleaning new cells, and moving accordingly. It soon finds it can move to `(2,1)` and cleans it.
   - At `(2,1)`, the robot cannot move right or down because of walls; it moves up and finds it's already visited `(1,1)`. After rotating right, it can move left to `(2,0)` but since it's been visited, the robot rotates right again and finally faces the last unexplored direction at `(2,1)` which is the cell on its left `(2,2)`.
   - At `(2,2)`, which is the last cell, the robot can't move up or right due to walls, and the left and down directions lead to already cleaned cells. It has now finished cleaning.

By the end of this process, the robot has covered the entire accessible area, and each `1` has been visited and turned into a `2` (indicating the robot has cleaned it).

Resulting visited cells:

```
1. 2 2 2
2. 2 2 2
3. 2 2 2
```

This example shows how the robot effectively uses the DFS method to clean the entire room. Each step is guided by whether the space has been visited or if there's an obstacle, and the visited set `vis` ensures that no space gets cleaned twice. The right turning mechanism helps the robot to systematically explore all four directions in its current location.

Python Solution

```python
class Solution:
    def cleanRoom(self, robot):
        """
        Cleans the entire room using a depth-first search algorithm.

        :type robot: Robot
        :rtype: None
        """

        def go_back():
            """
            Makes the robot go back to the previous cell and restore the original direction.
            """
            robot.turnLeft()
            robot.turnLeft()  # Rotate 180 degrees to face the opposite direction
            robot.move()
            robot.turnLeft()
            robot.turnLeft()  # Rotate another 180 degrees to restore initial direction

        def dfs(x, y, direction):
            """
            Clean the room recursively using depth-first search.

            :param x: Current x-coordinate of the robot
            :param y: Current y-coordinate of the robot
            :param direction: Current direction the robot is facing
            """
            visited.add((x, y))
            robot.clean()

            # Loop through all directions: 0 - up, 1 - right, 2 - down, 3 - left
            for k in range(4):
                new_direction = (direction + k) % 4
                new_x = x + directions[new_direction][0]
                new_y = y + directions[new_direction + 1]

                if (new_x, new_y) not in visited and robot.move():
                    dfs(new_x, new_y, new_direction)
                    go_back()  # Go back to the previous cell after cleaning

                # Turn the robot clockwise to explore next direction
                robot.turnRight()

        # Define directions corresponding to up, right, down, left movements
        # in order: up(-x, 0), right(0, 1), down(1, 0), left(0, -1)
        directions = [-1, 0, 1, 0, -1]

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

        # Start the DFS from the starting point (0,0) facing up (direction 0)
        dfs(0, 0, 0)
```

Java Solution

```java
import java.util.Set;
import java.util.HashSet;
import java.util.List;

// We're assuming that the Robot interface is defined elsewhere according to the initial code block
// provided. Robot
interface Robot {
    public void move();
    public void turnLeft();
    public void turnRight();
    public void clean();
}

public class RobotCleaner {
    // Store the four possible directions the robot can move: up, right, down, left.
    private final int[] directions = {-1, 0, 1, 0, -1};

    // Create a HashSet to keep track of visited cells, represented as (x, y) coordinates.
    private final Set<List<Integer>> visited = new HashSet<>();

    // The robot interface instance.
    private Robot robot;

    /**
     * Public method to clean the room. This method will be called initially.
     *
     * @param robot The robot interface through which we control and clean.
     */
    public void cleanRoom(Robot robot) {
        this.robot = robot;
        // Begin the depth-first search (DFS) at the starting point (0, 0) with an initial direction (0).
        dfs(0, 0, 0);
    }

    /**
     * Perform DFS to clean the room. Will explore all four directions at each point.
     *
     * @param row       Current row position of the robot.
     * @param col       Current column position of the robot.
     * @param direction Current direction the robot is facing.
     *                  (0, 1, 2, 3) corresponds to (up, right, down, left);
     */
    private void dfs(int row, int col, int direction) {
        // Clean the current cell.
        robot.clean();

        // Mark the current cell as visited.
        visited.add(List.of(row, col));

        // Explore all four directions.
        for (int k = 0; k < 4; ++k) {
            // Compute the new direction.
            int newDirection = (direction + k) % 4;

            // Compute the next cell's coordinates based on the new direction.
            int nextRow = row + directions[newDirection];
            int nextCol = col + directions[newDirection + 1];

            // If the next cell has not been visited and is accessible, move and continue DFS.
            if (!visited.contains(List.of(nextRow, nextCol)) && robot.move()) {
                dfs(nextRow, nextCol, newDirection);

                // Move back to the previous cell (backtracking).
                robot.turnRight();
                robot.turnRight();
                robot.move();

                // Restore to the original direction before the next loop iteration.
                robot.turnRight();
                robot.turnRight();
            }

            // Turn the robot to the next direction (90 degrees to the right).
            robot.turnRight();
        }
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    void cleanRoom(Robot& robot) {
        // Directions representing the four possible movements: up, right, down, left
        int directions[5] = {-1, 0, 1, 0, -1};
        set<pair<int, int>> visited;

        // Depth-first search algorithm to traverse and clean the room recursively
        function<void(int, int, int)> dfs = [&](int x, int y, int dir) {
            // Mark the current cell as visited
            robot.clean();
            // Mark the current cell as visited
            visited.insert({x, y});

            // Explore all four directions
            for (int k = 0; k < 4; ++k) {
                // Calculate the new direction after turning
                int newDir = (dir + k) % 4;
                // Calculate the coordinates of the adjacent cell
                int nx = x + directions[newDir], ny = y + directions[newDir + 1];

                // If the adjacent cell has not been visited and we can move there,
                // we move, clean the cell, and backtrack after
                if (!visited.count({nx, ny}) == 0 && robot.move()) {
                    dfs(nx, ny, newDir);

                    // Backtrack to the previous cell (requires two turns and move)
                    robot.turnRight();
                    robot.turnRight();
                    robot.move();
                    robot.turnRight();
                    robot.turnRight();
                }

                // Turn the robot to face the next direction
                robot.turnRight();
            }
        };

        // Start the cleaning process from (0, 0) facing up (direction index 0)
        dfs(0, 0, 0);
    }
};
```

Typescript Solution

```typescript
// Directions for the robot to move: up, right, down, and left
const directions: number[] = [-1, 0, 1, 0, -1];

// Set to keep track of visited cells to avoid cleaning the same cell multiple times
const visited: Set<string> = new Set<string>();

// Recursive depth-first search (DFS) function to clean the room
function dfs(x: number, y: number, currentDir: number, robot: Robot) {
    // Mark the current cell as visited by adding it to the visited set
    visited.add(`${x},${y}`);
    // Clean the current cell
    robot.clean();

    // Explore all four directions: up, right, down, and left
    for (let k = 0; k < 4; ++k) {
        // Calculate new direction after turning right 'k' times from current direction
        const newDir = (currentDir + k) % 4;
        // Get the new cell coordinates based on the direction
        const newX = x + directions[newDir];
        const newY = y + directions[newDir + 1];

        // If the new cell has not been visited and is not blocked
        if (!visited.has(`${newX},${newY}`) && robot.move()) {
            // Recursively clean the new cell
            dfs(newX, newY, newDir, robot);

            // Backtrack to the previous cell, facing the same direction as before
            robot.turnRight();
            robot.turnRight();
            robot.move();
            robot.turnRight();
            robot.turnRight();
        }

        // Turn the robot to the right (next direction)
        robot.turnRight();
    }
}

function cleanRoom(robot: Robot) {
    // Start DFS from the initial cell (0, 0) facing up (direction index 0)
    dfs(0, 0, 0, robot);
}
```

Time and Space Complexity

The time complexity of the above code is $O(4^2(m*n))$, where m is the total number of cells in the room and n is the number of obstacles. This is because the algorithm has to visit each non-obstacle cell once and at each cell, it makes up to 4 decisions – move in 4 possible directions. The recursion may go up to 4 branches at each level but would not revisit cells that are already visited, summarized by visited set `vis`.

The space complexity of the DFS is $O(n)$ for the recursive call stack as well as the space to hold the set of visited cells (in the worst case where there are no obstacles and we can move to every cell). However, in a densely packed room with obstructions, the number of visited states will be less than n.