1034. Coloring A Border

Medium Depth-First Search Breadth-First Search Array Matrix

Leetcode Link

Problem Description

You are given a matrix of integers where each integer represents a color. You need to identify a connected component in the matrix based on a starting cell given by its row and column indexes. A connected component entails all adjacent cells (up, down, left, or right) with the same color. The border of this connected component includes all cells in the component adjacent to cells with different colors or cells lying on the grid's boundary.

Your task is to recolor only the cells on the border of this connected component with a new color. The index of the starting cell within the connected component and the new color are given. The updated grid should be returned, showing the new border color while leaving the interior of the connected component and the rest of the grid unchanged.

Intuition

The solution to this problem uses Depth-First Search (DFS), an algorithm well-suited for exploring and marking the structure of a connected component within a grid. We begin at a cell `grid[row][col]` and look for all adjacent cells that share the same color, which signifies that they are part of the same connected component.

The intuition behind using DFS is to traverse through all the cells in the connected component starting from the specified cell `grid[row][col]`. As we do this, we need to determine which cells are on the border of the component. A border cell is defined as a cell that:

1. Is on the edge of the grid, or
2. Is adjacent to a cell that has a different color than the one we started with.

The DFS function checks all four directions around a cell. For every cell that we visit:

- If the cell is within the grid and the adjacent cell is not visited yet, we check:
  - If the adjacent cell has the same color, we continue DFS on that cell since it's part of the same component.
  - If the adjacent cell has a different color, we know that we are on the border, and thus we change the color of the current cell.
- If the cell is on the edge of the grid, it's a border cell, so we change the color of the current cell.

We iterate in this manner until all the cells in the connected component have been visited and the appropriate border cells have been recolored. An auxiliary `vis` matrix is used to keep track of visited cells to avoid processing a cell more than once, which helps in preventing an infinite loop.

Finally, we return the modified grid with the newly colored borders, while the interior of the connected component and all other parts of the grid remain unchanged.

Solution Approach

The solution utilizes a **Depth-First Search (DFS)** approach to explore the grid and identify border cells of the connected component. The following are the key aspects of the implementation:

1. **DFS Method**: The solution defines a nested `dfs` function which is responsible for the recursive search. It accepts the current cell's indices `i` and `j`, as well as the original color `c` of the connected component.
2. **Tracking Visited Cells**: A two-dimensional list `vis` is created with the same dimensions as the input grid `grid`, initialized with `False` values. This list is used to keep track of cells that have already been visited to prevent infinite recursion and repeated processing.
3. **Checking Neighbors**: In each call to the `dfs` function, it iterates over the four adjacent cells (up, down, left, right) using a for-loop and the `pairwise` helper to generate pairs of directions.
4. **Edge Conditions**: The algorithm checks if a neighboring cell is within the boundaries of the grid. If it is outside, it means the current cell is on the border of the grid and thus should be recolored.
5. **Coloring Border Cells**:
   - If the adjacent cell is within the grid and hasn't been visited (`vis[x][y]` is `False`), there are two conditions:
     - If the adjacent cell has the same color as the original (`grid[x][y] == c`), the `dfs` function is called recursively for that cell to continue the component traversal.
     - If the adjacent cell has a different color, the current cell is on the border of the connected component and its color is changed to `color`.
6. **Calling DFS**: After initializing the `vis` grid, the `dfs` function is invoked starting from the cell `grid[row][col]`, using its color as the reference for the connected component.
7. **Returning the Result**: Once the DFS traversal is complete and all border cells are recolored, the modified grid is returned.

The implementation creatively uses recursion to deeply explore the connected component and the auxiliary `vis` matrix to manage state across recursive calls. The determination of border cells is carried out in a granular fashion by checking each neighbor, which effectively allows the coloring to occur only where it is required.

Example Walkthrough

Let's consider a small 4×4 matrix and walk through the solution approach with a given starting cell and new color:

```
Grid:
1 1 1 2 3 4
2 1 1 2 3 4
2 1 1 2 3 4
4 5 5 5 4 4

Starting Cell: (1, 1) (Zero-indexed, meaning the cell in row 1, column 1)
Original Color of Starting Cell: 1
New Border Color: 9
```

First, we initialize an auxiliary 2D list `vis` of the same size as the grid with `False` values to keep track of whether a cell has been visited. Next, we define a `dfs` function to explore the grid starting from the given cell.

1. We call the `dfs` function on the starting cell `(1, 1)` with its color `1`. Since it matches the original color, we continue with DFS. The `vis` grid will be updated with `True` at this cell to denote it's been visited.
2. The `dfs` function now checks adjacent cells `(1, 0)`, `(1, 2)`, `(0, 1)`, and `(2, 1)`.
3. Whenever visiting an adjacent cell, the border conditions are checked to ensure we are within the grid. Cells at `(1, 0)` and `(1, 2)` have the same color as the original and are within the grid; they're not border cells, and the `dfs` is called on them.
4. Cells `(0, 1)` and `(2, 1)` are also part of the connected component, so DFS continues. Meanwhile, `vis` is continuously updated.
5. By the time we check `(0, 1)`, we find that the cell `(0, 0)` is of a different color. This means `(0, 1)` is a border cell because it's adjacent to a differently colored cell.
6. The cells on the actual edge of the matrix are by default border cells, hence their colors are changed to the new border color `9`.
7. As we continue this process recursively, all border cells are identified and changed. For this example, the border consists of cells `(1, 0)`, `(0, 1)`, `(1, 1)`, `(1, 2)`, and `(2, 1)`.
8. After all recursion steps complete, the grid is updated as follows:

```
1 1 9 2 3 4
2 9 1 2 3 4
2 9 9 2 3 4
4 5 5 5 4 4
```

9. The `dfs` call terminates, and we return the modified grid with the borders recolored to `9`.

The steps in the `dfs` ensure that all cells that are in the connected component of the original color are explored, but only the actual border cells are recolored as per the conditions. This is effective at preserving the original structure of the component while updating the border as specified.

Python Solution

```python
from typing import List

class Solution:
    def colorBorder(
        self, grid: List[List[int]], row: int, col: int, color: int
    ) -> List[List[int]]:
        # Helper function to implement depth-first search
        def dfs(i: int, j: int, current_color: int) -> None:
            # Mark the cell as visited
            visited[i][j] = True
            # Iterate through the four possible directions up, right, down, left
            for direction in range(4):
                x, y = i + dirs[direction], j + dirs[direction + 1]
                # Check if x and y are within the grid boundaries
                if 0 <= x < rows and 0 <= y < cols:
                    # If the neighbor has not been visited and has the same color, perform dfs on it
                    if not visited[x][y]:
                        if grid[x][y] == current_color:
                            dfs(x, y, current_color)
                        else:
                            # If the neighbor is of a different color, color the border
                            grid[i][j] = color
                # Else
                else:
                    # If the cell is at the boundary, also color the border
                    grid[i][j] = color

        # Number of rows and columns in the grid
        rows, cols = len(grid), len(grid[0])
        # Initialize a 2D list to track visited cells
        visited = [[False] * cols for _ in range(rows)]
        # Directions for up, right, down, left
        dirs = [-1, 0, 1, 0, -1]
        dfs(r, c, grid[r][c])
        # Start the depth-first search from the given row and col
        # Return the colored grid
        return grid
```

Java Solution

```java
class Solution {
    private int[][] grid; // Represents the grid of colors.
    private boolean[][] visited; // The color to be used for the border.
    private int rows; // Number of rows in the grid.
    private int cols; // Number of columns in the grid.
    private boolean[][] visited; // To keep track of visited cells during DFS.

    public int[][] colorBorder(int[][] grid, int row, int col, int color) {
        this.grid = grid;
        this.newColor = color;
        rows = grid.length;
        cols = grid[0].length;
        visited = new boolean[rows][cols];

        // Perform DFS starting from the given cell.
        dfs(row, col, grid[row][col]);
        return grid;
    }

    private void dfs(int i, int j, int originalColor) {
        visited[i][j] = true; // Mark the current cell as visited.
        int[] directions = {-1, 0, 1, 0, -1}; // To traverse in the 4 possible directions: up, right, down, left.

        // Check if the current cell is a border cell and should be colored.
        boolean isBorderCell = i == 0 || i == rows - 1 || j == 0 || j == cols - 1;

        for (int k = 0; k < 4; ++k) {
            int x = i + directions[k], y = j + directions[k + 1];
            if (x >= 0 && x < rows && y >= 0 && y < cols) {
                // If the neighboring cell has not been visited and has the original color, continue DFS.
                if (!visited[x][y]) {
                    if (grid[x][y] == originalColor) {
                        dfs(x, y, originalColor);
                    } else {
                        // If the neighboring cell has a different color, this is a border cell.
                        isBorderCell = true;
                    }
                }
            } else {
                // If the cell is on the grid edge, it is a border cell.
                isBorderCell = true;
            }
        }

        // If the current cell is a border cell, color it with the new color.
        if (isBorderCell) {
            grid[i][j] = newColor;
        }
    }
}
```

C++ Solution

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

using namespace std;

class Solution {
public:
    vector<vector<int>> colorBorder(vector<vector<int>>& grid, int row, int col, int color) {
        int numRows = grid.size(); // number of rows in the grid
        int numCols = grid[0].size(); // number of columns in the grid

        // Define a visited matrix to track visited cells
        bool visitedFromRec[numRows][numCols];
        memset(visited, false, sizeof(visited));

        // Direction arrays for traversing the four neighbors of a cell (up, right, down, left)
        int directions[5] = {-1, 0, 1, 0, -1};

        // Recursive Depth-First Search (DFS) lambda function
        function<void(int, int, int)> dfs = [&](int i, int j, int originalColor) {
            visited[i][j] = true; // Mark the current cell as visited
            for (int k = 0; k < 4; ++k) { // Loop through each neighbor
                int nextRow = i + directions[k];
                int nextCol = j + directions[k + 1];
                if (nextRow >= 0 && nextRow < numRows && nextCol >= 0 && nextCol < numCols) {
                    if (!visitedFromRec[nextRow][nextCol]) { // If neighbor has not been visited
                        if (grid[nextRow][nextCol] == originalColor) { // If neighbor has the same color
                            dfs(nextRow, nextCol, originalColor);
                        } else {
                            // If color is different, color the border
                            grid[i][j] = color;
                        }
                    }
                } else {
                    // If it's the border of the grid, color it
                    grid[i][j] = color;
                }
            }
        };

        // Start DFS from the given cell (row, col)
        dfs(row, col, grid[row][col]);

        // Return the updated grid
        return grid;
    }
};
```

Typescript Solution

```typescript
// Function to color the border of a connected component in the grid.
function colorBorder(grid: number[][], row: number, col: number, newColor: number): number[][] {
    const rowCount = grid.length; // Number of rows in the grid.
    const colCount = grid[0].length; // Number of columns in the grid.
    const visited = new Array(rowCount).fill(0).map(() => new Array(colCount).fill(false)); // Visited cells tracker.

    // Directions for traversing up, right, down, and left.
    const directions = [-1, 0, 1, 0, -1];

    // Depth-First Search to find and color the border.
    function depthFirstSearch(row: number, col: number, originalColor: number): void {
        visited[row][col] = true; // Mark the current cell as visited.

        // Explore all four directions from the current cell.
        for (let k = 0; k < 4; ++k) {
            const nextRow = row + directions[k];
            const nextCol = col + directions[k + 1];

            if (nextRow >= 0 && nextRow < rowCount && nextCol >= 0 && nextCol < colCount) {
                if (!visited[nextRow][nextCol]) {
                    if (grid[nextRow][nextCol] === originalColor) {
                        // Continue the DFS if the next cell has the same color.
                        depthFirstSearch(nextRow, nextCol, originalColor);
                    } else {
                        // Color the current cell if the next cell has a different color.
                        grid[row][col] = newColor;
                    }
                }
            } else {
                // Color the current cell if it is on the border of the grid.
                grid[row][col] = newColor;
            }
        }
    }

    // Color the connected component's border starting from the given cell.
    depthFirstSearch(row, col, grid[row][col]);

    // Apply the new color to any cell that is marked as visited but not colored yet, indicating it's on the border.
    for (let i = 0; i < rowCount; ++i) {
        for (let j = 0; j < colCount; ++j) {
            // A cell is on the border if it is visited and has at least one neighboring cell with a different color or be at the grid's edge
            if (visited[i][j] && (i === 0 || i === rowCount - 1 || j === 0 || j === colCount - 1 ||
                (grid[i - 1][j] !== grid[i][j]) ||
                (grid[i + 1][j] !== grid[i][j]) ||
                (grid[i][j - 1] !== grid[i][j]) ||
                (grid[i][j + 1] !== grid[i][j]))) {
                grid[i][j] = newColor;
            }
        }
    }

    // Return the modified grid.
    return grid;
}
```

Time and Space Complexity

The provided Python code performs a depth-first search (DFS) to color the borders of a connected component in a grid based on the given conditions. The complexity analysis is as follows:

Time Complexity:

The time complexity of this DFS algorithm is mainly determined by the number of cells in the grid that are visited. In the worst case, the algorithm will visit every cell in the grid once. There are $ m * n $ cells, where $ m $ is the number of rows and $ n $ is the number of columns in the grid. Therefore, the DFS will have a time complexity of $ O(m * n) $.

Space Complexity:

The space complexity includes the memory taken by the `vis` matrix to keep track of visited cells, as well as the recursion call stack for DFS. The `vis` matrix has the same dimensions as the input grid, so it takes $ O(m * n) $ space.

The recursion depth can go as deep as the maximum size of the connected component in the grid. In the worst case, this could be the entire grid, which would mean that the call stack can potentially grow up to $ m * n $ calls deep. Hence, the space complexity due to the recursion stack could also be $ O(m * n) $.

Therefore, the overall space complexity is $ O(m * n) $ as well.

Note: The provided code contains a `pairwise` function which is not defined within the code snippet or as part of the standard Python library up to the knowledge cutoff date. Assuming it's intended to generate pairs of coordinates for traversing adjacent cells, its behavior seems to be that of incorrectly iterating over the grid coordinates without actually yielding pairs.