733. Flood Fill

Easy Depth-First Search Breadth-First Search Array Matrix Leetcode Link

Problem Description

In this problem, we have a grid that represents an image, where each element in the grid is an integer that symbolizes the pixel value of the image. We're given a starting pixel defined by its row number `sr` and column number `sc`, and a new color value `color`. The task is to perform a "flood fill" on the image beginning from the starting pixel.

A flood fill is akin to pouring paint into a single spot and watching it spread out, coloring all connected areas with the same initial color. The fill spreads only in four directions: up, down, left, and right, from any given pixel. It continues to spread until it encounters a pixel with a different color than the starting pixel's original color.

The objective is to replace the color of the starting pixel and all connected pixels (4-directionally) with the same original color as the starting pixel with the new `color`. The end result is the image with the modified colors, which we should return.

Intuition

The intuitive approach to solving this problem is to simulate the flooding process using either Depth-First Search (DFS) or Breadth-First Search (BFS). Both are traversal algorithms that can navigate through the grid to find all pixels that are eligible for recoloring.

The intuition behind DFS is to start at the given pixel (sr, sc), change its color to the new color, and then recursively change the color of all adjacent pixels (up, down, left, right) that are the same color as the initial one. This does mean checking whether the new pixel position is within the boundary of the image and whether the pixel color matches the original color of the starting pixel.

For this solution, we'll use DFS:

1. From the starting pixel, update its color to `color`.
2. Look at adjacent pixels. For each adjacent pixel:
   - If the pixel is within bounds, and
   - If the pixel's color is the same as the starting pixel's original color, and
   - the pixel isn't already the new color (to prevent infinite recursion),
   - then recursively apply the flood fill to that pixel.

The recursive function spreads out from the initial pixel until it has reached all connected pixels with the original color, effectively performing the flood fill and updating the image as required.

Solution Approach

The solution employs the Depth-First Search (DFS) algorithm, and the approach can be outlined in the following steps:

1. **Initial setup**: We define a `dfs` helper function that will be used to perform the depth-first traversal and paint the image. The image matrix's dimensions are stored in `m` and `n` for convenience in bounds checking. The original color (`oc`) is recorded—it's the color of the starting pixel (at `sr`, `sc`).

2. **Defining directions**: A tuple named `dirs` is defined, which includes the relative positions that represent the 4-directional moves: up, down, left, and right. The tuple has the pattern `(-1, 0, 1, 0, -1)` in order to easily loop through pairs of directions for up-down-left-right traversal.

3. **Recursion with DFS**: The `dfs` function takes a position `(i, j)`:

   - It checks if the current position is out of bounds or if the pixel's color is not the same as the original color (`oc`) or is already painted with the new color—this is the base case for the recursion to stop or prevent infinite loops.
   - If the base case does not hold, the current pixel's color is changed to the `color`.
   - For each 4-directional neighbor (calculated using the `dirs` tuple), apply DFS recursion by calling `dfs` on the neighbor's coordinates.

```python
def dfs(i, j):
    if (
        not 0 <= i < m
        or not 0 <= j < n
        or image[i][j] != oc
        or image[i][j] == color
    ):
        return
    image[i][j] = color
    for a, b in pairwise(dirs):
        dfs(i + a, j + b)
```

   In the implementation, `pairwise(dirs)` construct is used (assuming it's similar to Python's `pairwise` utility to make pairs from the `dirs` tuple), which should create pairs of directions we want to move in (up, down, left, and right). There is a detail here: most Python environments do not have a built-in function called `pairwise`. Hence, it's likely meant to be a custom utility defined somewhere else, or an import from some module like `itertools`. For example, if it was intended as `itertools.pairwise()`, we would need to import itertools and make sure to handle the last repetitive direction.

4. **Initiating the fill**: The DFS recursion is initiated by calling `dfs(sr, sc)`.

5. **Returning the result**: After DFS completes, it returns the `image`, which by now has been flood-filled using the new color.

The DFS recursion ensures that all connected pixels of the same original color have been recolored, and as the recursion unwinds, the modified image is eventually returned.

Example Walkthrough

Let's consider a small grid as our example image, where we will apply the flood fill algorithm. Suppose the image is a 3×3 grid as follows:

```
1 1 2
1 1 0
1 0 3
```

The starting pixel for our flood fill is defined by the row number `sr = 1` and column number `sc = 1` (using 0-based indexing), and the new color we want to apply is `2`.

The pixel at (1, 1) has the color `1`. According to the flood fill algorithm, we want to change this color and all connected pixels of the same color to `2`.

Step 1: We start from the pixel at (1, 1). Its color is `1`, which matches the original color (hence eligible for color change). We change its color to `2`.

Our image now looks like this:

```
1 1 2
1 2 0
1 0 3
```

Step 2: We perform DFS from our starting pixel. We look at its four neighbors: up `(0, 1)`, down `(2, 1)`, left `(1, 0)`, and right `(1, 2)`.

- Pixel (0, 1) has color `1`, so it is eligible. We fill it with color `2`.
- Pixel (2, 1) has color `0` and is not eligible since it is not the original color.
- Pixel (1, 0) has color `1`, so it is eligible. We fill it with color `2`.
- Pixel (1, 2) has color `0` and is not eligible since it is not the original color.

Now our image looks like this:

```
1 2 2
2 2 0
1 0 3
```

Step 3: We now apply DFS on the neighbors of these newly colored pixels (0, 1) and (1, 0).

- Checking neighbors of (0, 1): The up and down directions are out of bounds, the left pixel (0, 0) with color `1` gets filled, and the right pixel (0, 2) has color `2`.

- Checking neighbors of (1, 0): All sides except the left pixel (1, -1), which is out of bounds, have already been checked or don't have the original color.

Our final image now looks like this:

```
2 2 2
2 2 0
1 0 3
```

The image has been successfully filled using the new color `2`, starting from the pixel at (1, 1). Pixels connected with the same original color have been recolored, illustrating how the flood fill algorithm works.

Python Solution

```python
from typing import List

class Solution:
    def floodFill(self, image: List[List[int]], start_row: int, start_col: int, new_color: int) -> List[List[int]]:
        """
        Perform a flood fill on the image starting from the pixel at (start_row, start_col), changing all
        connected pixels of the same color to the new color.

        :param image: 2-D array representing the image.
        :param start_row: The row of the starting pixel.
        :param start_col: The column of the starting pixel.
        :param new_color: The new color to apply to the connected pixels.
        :return: The image after the flood fill operation.
        """
        def flood_fill_helper(row: int, col: int):
            # If the pixel is out of bounds, not the original color, or already the new color,
            # then do not proceed with the fill
            if (not 0 <= row < height or not 0 <= col < width or
                    image[row][col] != original_color or image[row][col] == new_color):
                return

            # Change the pixel to the new color
            image[row][col] = new_color

            # Visit all four adjacent pixels (up, right, down, left)
            for delta_row, delta_col in zip(directions[:-1], directions[1:]):
                flood_fill_helper(row + delta_row, col + delta_col)

        # Define the directions for moving to adjacent pixels
        directions = [-1, 0, 1, 0, -1]
        # Get the dimensions of the image
        height, width = len(image), len(image[0])
        # Get the original color of the starting pixel
        original_color = image[start_row][start_col]

        # Begin the flood fill from the starting pixel
        flood_fill_helper(start_row, start_col)

        return image
```

Java Solution

```java
class Solution {

    // Direction vectors representing the 4 connected pixels (up, right, down, left).
    private int[] directions = {-1, 0, 1, 0, -1};
    // The image we need to modify.
    private int[][] image;
    // The new color to apply to the flood fill.
    private int newColor;
    // The original color to be replaced.
    private int originalColor;

    // Method to begin flood fill operation
    public int[][] floodFill(int[][] image, int startRow, int startColumn, int color) {
        // Initialize the image, new color, and original color based on the input.
        this.image = image;
        this.newColor = color;
        this.originalColor = image[startRow][startColumn];

        // Call the recursive dfs method starting from the pixel at (sr, sc)
        dfs(startRow, startColumn);
        // Return the modified image after the flood fill operation.
        return image;
    }

    // Depth-first search (DFS) method to apply new color to connected components.
    private void dfs(int row, int column) {
        // Boundary check: if the pixel is out of bounds or isn't the original color or is already the new color, return.
        if (row < 0 || row >= image.length || column < 0 || column >= image[0].length ||
            image[row][column] != originalColor || image[row][column] == newColor) {
            return;
        }

        // Change the color of the current pixel to the new color.
        image[row][column] = newColor;

        // Iterate through each of the 4 connected neighbors.
        for (int k = 0; k < 4; ++k) {
            // Recursively call dfs for the current neighbor.
            dfs(row + directions[k], column + directions[k + 1]);
        }
    }
}
```

C++ Solution

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

class Solution {
public:
    // Method to perform flood fill algorithm.
    vector<vector<int>> floodFill(vector<vector<int>>& image, int startRow, int startCol, int newColor) {
        int rowCount = image.size();
        int colCount = image[0].size();
        int oldColor = image[startRow][startCol];

        // Directions to move in the matrix - up, right, down, left
        int directions[5] = {-1, 0, 1, 0, -1};

        // Define depth-first search lambda function to apply the new color recursively
        std::function<void(int, int)> dfs = [&](int row, int col) {
            // Check for out-of-bounds, if the color is different from the oldColor, or if it's already filled with the newColor
            if (row < 0 || row >= rowCount || col < 0 || col >= colCount || image[row][col] != oldColor || image[row][col] == newColo
                return;
            }
            // Apply the new color
            image[row][col] = newColor;
            // Perform DFS in all directions
            for (int k = 0; k < 4; ++k) {
                dfs(row + directions[k], col + directions[k + 1]);
            }
        };

        // Start the flood fill from the (startRow, startCol)
        dfs(startRow, startCol);
        return image; // Return the modified image
    }
};
```

Typescript Solution

```typescript
/**
 * Performs a flood fill on an image starting from the pixel at (sr, sc).
 *
 * @param {number[][]} image - The 2D array of numbers representing the image.
 * @param {number} sr - The row index of the starting pixel.
 * @param {number} sc - The column index of the starting pixel.
 * @param {number} newColor - The color to fill with.
 * @returns {number[][]} - The modified image after performing the flood fill.
 */
function floodFill(image: number[][], sr: number, sc: number, newColor: number): number[][] {
    // Determine the dimensions of the image
    const rowCount = image.length;
    const colCount = image[0].length;

    // Target color to replace
    const targetColor = image[sr][sc];

    /**
     * Recursive depth-first search function to perform the color fill.
     *
     * @param {number} row - The current row index.
     * @param {number} col - The current column index.
     */
    function dfs(row: number, col: number): void {
        // Check if the current pixel is outside the image boundaries,
        // already has the new color, or is not matching the target color.
        if (
            row < 0 || row === rowCount ||
            col < 0 || col === colCount ||
            image[row][col] !== targetColor ||
            image[row][col] === newColor
        ) {
            // Exit the function without further processing.
            return;
        }

        // Fill the current pixel with the new color.
        image[row][col] = newColor;

        // Recursively apply the fill operation to the neighboring pixels.
        dfs(row + 1, col); // Down
        dfs(row - 1, col); // Up
        dfs(row, col + 1); // Right
        dfs(row, col - 1); // Left
    }

    // Call the dfs function on the starting pixel.
    dfs(sr, sc);

    // Return the updated image after the flood fill.
    return image;
}
```

Time and Space Complexity

The **time complexity** of the flood fill algorithm is `O(M*N)`, where `M` is the number of rows and `N` is the number of columns in the image. This is because in the worst case, the algorithm performs a depth-first search (DFS) on every cell in the grid once when the entire image requires to be filled with the new color.

The **space complexity** is also `O(M*N)`, primarily due to the recursive stack that could potentially grow to the size of the entire grid in the case of a large connected region with the same color that needs to be filled.