2556. Disconnect Path in a Binary Matrix by at Most One Flip

Medium | Depth-First Search | Breadth-First Search | Array | Dynamic Programming | Matrix | Leetcode Link

Problem Description

The problem presents us with an `m x n` binary matrix, where `0` and `1` represent blocked and open cells, respectively. From any cell, `(row, col)`, we can move to an adjacent cell to the right `(row, col + 1)` or below `(row + 1, col)` providing it contains a `1`.

The task is to determine whether we can make the matrix "disconnected" by flipping at most one cell's value, except for the cells at `(0, 0)` and `(m - 1, n - 1)`. A disconnected matrix means there is no path from the start `(0, 0)` to the end `(m - 1, n - 1)`.

Flipping a cell switches its value: if the cell was `1`, it becomes `0`, and if it was `0`, it becomes `1`. The challenge lies in deciding if it's possible to disrupt any existing path with a single flip, assuming such a path exists.

Intuition

The key intuition for this solution is to perform a Depth-First Search (DFS) from the starting cell `(0, 0)` two times.

Initially, we execute DFS to check if there exists a path from `(0, 0)` to `(m - 1, n - 1)` without flipping any cells. If such a path exists, then we proceed to the next step. Otherwise, if there is no such path, the grid is already disconnected, and we can return `true` early (which the code doesn't handle explicitly).

After conducting the initial DFS, we mark the beginning and ending cells as open (if they aren't already), hoping to test whether forcing these cells open and disconnecting another cell can lead to a disconnected path. Therefore, another DFS is executed with these conditions.

Here's the process:

1. Start by marking the current cell as visited (flipping from `1` to `0` to avoid revisiting).
2. If the end cell `(m - 1, n - 1)` is reached, we've found a path.
3. If we cannot find a path to the end cell, mark it as not possible to disconnect by only flipping one cell.
4. Perform the first DFS to check for the existing path.
5. If a path is found in step 4, flip the start and end cells to `1` (to ensure they are considered in the next DFS), then conduct the second DFS, enforcing the use of at least one flip elsewhere.
6. Lastly, assess the outcomes from both DFS searches. If both searches find a path, we conclude that it is not possible to disconnect the grid with a single flip. If the second DFS does not find a path, it shows that flipping at most one cell (besides the start and end) can disconnect the grid. We return `true` if the grid can be disconnected, `false` otherwise.

This approach works because flipping the cells to `1` after the initial search guarantees that if there is another path aside from the one found during the first traversal, the second traversal will reveal whether blocking one additional cell leads to disconnection.

Solution Approach

The solution leverages the Depth-First Search (DFS) algorithm, which is a common technique for exploring all the paths in a grid-like structure or graph. It works by moving from the starting node and venturing as far as possible along each branch before backtracking.

In the Python code provided, a nested function `dfs(i, j)` is defined for the DFS traversal. This function aims to determine whether a path exists from the current cell `(i, j)` to the bottom-right cell `(m - 1, n - 1)`.

Here's a breakdown of the DFS implementation:

1. **Base Case**: If the current cell indices are out of bounds (greater than or equal to `m` or `n`), or if the current cell value is `0` (blocked), DFS returns `False`, indicating no path can be made through this cell.

2. **Visiting Cells**: When a cell with a value `1` is visited, the function first marks it as visited by flipping it to `0`. This prevents the DFS from revisiting the same cell and getting stuck in a loop.

3. **Path Completion Check**: If the end cell `(m - 1, n - 1)` is reached, DFS returns `True`, confirming a path has been found.

4. **Exploring Adjacent Cells**: The DFS continues by making recursive calls to the cell directly below `(i + 1, j)` and to the cell directly to the right `(i, j + 1)`. If any of those recursive calls return `True`, the current function also returns `True`.

5. **Determining Disconnectability**:

    - Conduct the first DFS from the start cell `(0, 0)` using `a = dfs(0, 0)` to check for an existing path.
    - After the initial DFS, enforce the start and end cells to be open by setting `grid[0][0]` and `grid[-1][-1]` to `1`.
    - Perform a second DFS with the modified grid using `b = dfs(0, 0)`.
6. **Result Evaluation**: The piece of code `not (a and b)` returns `True` only if the first DFS found a path (`a` is `True`), and the second DFS did not find a path (`b` is `False`). Thus, the expression evaluates to `True` if we can disconnect the matrix, otherwise, it returns `False`.

The effectiveness of the solution is due to the algorithm's ability to explore all possible paths from the start to end cell, and the logical use of DFS to determine if disrupting any cell's connectivity (by changing a single `1` to `0`) will disconnect the grid without testing every single cell explicitly. The solution cleverly checks the disconnectability of the matrix by attempting to find two distinct paths before and after assuming at least one flip elsewhere in the grid.

Example Walkthrough

Let's consider a simple `3 x 3` binary matrix example to illustrate the solution approach:

```
Initial Grid:
1 1 1
0 1 0
1 0 1
```

Following the solution steps:

1. We attempt to perform DFS from the start cell `(0, 0)`. We check for the path existence without any flip and mark the visited cells.
2. Since we start at `(0, 0)`, we mark it as `0`, to not revisit and move right to `(0, 1)` and then to `(0, 2)`.
    - Now in the top right corner, `(0, 2)`, we go down to `(1, 2)` but find that it's blocked (`0`).
    - We backtrack and from `(0, 2)` and move down to `(1, 2)` and find it is also blocked.
    - Backtracking again, we move down from `(0, 1)` to `(1, 1)` and since it has a `1`, we continue by going right to `(1, 2)` which is blocked.
    - We go down from `(1, 1)` to `(2, 1)` but it is blocked.
    - Backtracking again, we go down from `(0, 0)` straight to `(1, 0)` which is blocked.
    - There is no more path to explore and we couldn't reach `(2, 2)`, the end cell.

```
Grid after first DFS attempt:
0 0 1
0 1 0
1 0 1
```

3. Since the first DFS didn't find any path, we know that the grid is already disconnected.
4. There is no need to proceed with the second DFS in this case since the grid is disconnected after the first DFS. We can directly return `true`.

Given the grid is already disconnected, this example demonstrates the scenario where the initial DFS itself concludes that no path exists from start to end without flipping any cells, thus no need for a second DFS or flipping any additional cells to conclude that the grid can be disconnected by at most one flip.

Python Solution

```python
class Solution:
    def isPossibleToCutPath(self, grid: List[List[int]]) -> bool:
        # dfs is a helper function that performs a depth-first search on the grid to
        # determine if there is a path from the top-left corner (0,0) to the bottom-right corner (m-1, n-1)
        def dfs(row, col):
            # Base case: if we are out of bounds or the current cell is 0 (a wall), return False.
            if row >= rows or col >= cols or grid[row][col] == 0:
                return False
            # Mark the current cell as visited by changing it to 0 (a wall).
            grid[row][col] = 0
            # If we have reached the bottom-right corner, return True.
            if row == rows - 1 and col == cols - 1:
                return True
            # Recursively check the right and down directions from the current cell.
            return dfs(row + 1, col) or dfs(row, col + 1)

        # Get the number of rows and columns in the grid.
        rows, cols = len(grid), len(grid[0])

        # Call dfs to see if there is a first path from top-left to bottom-right.
        has_path_first_time = dfs(0, 0)

        # Reset the top-left and bottom-right cells back to 1, as we made them 0 in the dfs.
        grid[0][0] = grid[-1][-1] = 1

        # Call dfs again to see if there is still a path after the first has been cut.
        has_path_second_time = dfs(0, 0)

        # If there was a path both times, that means cutting the first path did not
        # completely block off the grid, so we return False. Otherwise, we return True.
        return not (has_path_first_time and has_path_second_time)

```

Java Solution

```java
class Solution {
    private int[][] grid; // Store the provided grid
    private int rows; // Number of rows in the grid
    private int cols; // Number of columns in the grid

    // Determines if it's possible to cut a path from top-left to bottom-right
    public boolean isPossibleToCutPath(int[][] grid) {
        this.grid = grid;
        rows = grid.length; // Initialize number of rows
        cols = grid[0].length; // Initialize number of columns

        // Perform a DFS from the top-left corner before the cut
        boolean pathExistsBeforeCut = depthFirstSearch(0, 0);

        // Simulate the cut by marking the start and end points as obstacles
        grid[0][0] = 1;
        grid[rows - 1][cols - 1] = 1;

        // Perform a DFS from the top-left corner after making the cut
        boolean pathExistsAfterCut = depthFirstSearch(0, 0);

        // If the path exists before and not after, then it is possible to cut the path
        return !(pathExistsBeforeCut && pathExistsAfterCut);
    }

    // Helper method to perform depth-first search
    private boolean depthFirstSearch(int row, int col) {
        // Check boundaries and whether the current cell is an obstacle
        if (row >= rows || col >= cols || grid[row][col] == 0) {
            return false;
        }

        // Check if we've reached the bottom-right corner
        if (row == rows - 1 && col == cols - 1) {
            return true;
        }

        // Mark the current cell as an obstacle to avoid revisiting
        grid[row][col] = 0;

        // Recursively search in the right and down directions, returning true if a path is found
        return depthFirstSearch(row + 1, col) || depthFirstSearch(row, col + 1);
    }
}

```

C++ Solution

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

class Solution {
public:
    // The function evaluates if it's possible to block the path from top left to bottom right by setting a cell to zero
    bool isPossibleToCutPath(vector<vector<int>>& grid) {
        int rows = grid.size();            // Number of rows in the grid
        int cols = grid[0].size();         // Number of columns in the grid

        // Helper function to perform depth-first search from (row, col) position in the grid
        // Returns true if there is a path to bottom right, false otherwise.
        function<bool(int, int)> performDFS = [&](int row, int col) -> bool {
            // Base case checks: If out of bounds or encountered a zero, return false
            if (row >= rows || col >= cols || grid[row][col] == 0) {
                return false;
            }
            // Reached the bottom-right cell, return true
            if (row == rows - 1 && col == cols - 1) {
                return true;
            }
            // Set the current cell to 0 to prevent revisiting
            grid[row][col] = 0;

            // Explore the path downwards and rightwards;
            // the path exists if either of them returns true
            return performDFS(row + 1, col) || performDFS(row, col + 1);
        };

        // Check if there is a path from top left to bottom right in the current grid
        bool existsPathInitially = performDFS(0, 0);

        // Reset the top-left and bottom-right values of the grid to 1
        grid[0][0] = grid[rows - 1][cols - 1] = 1;

        // Check if there is a path after the first DFS and setting cells back to 1
        bool existsPathAfterReset = performDFS(0, 0);

        // If a path was found in both checks, then it's impossible to cut the path by setting a single cell to zero
        // Return the negation of the logical AND to indicate if it's possible to cut the path
        return !(existsPathInitially && existsPathAfterReset);
    }
};

```

Typescript Solution

```typescript
function isPossibleToCutPath(grid: number[][]): boolean {
    // Store the number of rows (m) and columns (n) from the input grid
    const numRows = grid.length;
    const numCols = grid[0].length;

    // Define the depth-first search function to traverse the grid
    const depthFirstSearch = (rowIndex: number, colIndex: number): boolean => {
        // Base case: return false if the current position is out of bounds or not a path (non-1 value)
        if (rowIndex >= numRows || colIndex >= numCols || grid[rowIndex][colIndex] !== 1) {
            return false;
        }
        // Mark the current cell as visited by setting it to 0
        grid[rowIndex][colIndex] = 0;

        // Base case: return true if reached the bottom-right cell of the grid
        if (rowIndex === numRows - 1 && colIndex === numCols - 1) {
            return true;
        }

        // Recursively explore the right and down neighbors of the current cell
        // Return true if any recursion call returns true
        return depthFirstSearch(rowIndex + 1, colIndex) || depthFirstSearch(rowIndex, colIndex + 1);
    };

    // Perform DFS traversal from the top-left corner (0,0)
    const firstTraversalResult = depthFirstSearch(0, 0);

    // Reset the top-left and bottom-right cells for the second traversal
    grid[0][0] = 1;
    grid[numRows - 1][numCols - 1] = 1;

    // Perform DFS traversal again to see if path can still be found after first pass
    const secondTraversalResult = depthFirstSearch(0, 0);

    // The path can be cut if both first and second traversals did not find a path (i.e., false status)
    return !(firstTraversalResult && secondTraversalResult);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given algorithm is primarily dependent on the depth-first search (DFS) implementation that explores all possible paths from the top-left to the bottom-right corner of the grid, if such a path exists. In the worst case, every cell in the grid might be visited during the DFS.

For a grid with `m` rows and `n` columns, the worst-case time complexity is `O(m*n)` since each cell is visited at most once during the search.

Space Complexity

The space complexity is primarily driven by the call stack used by the recursive DFS function. In the worst case, the function may recurse to a depth equal to the sum of the number of rows and columns of the grid, because at most, the path can be as long as moving to the rightmost column and then downwards to the bottommost row (or vice versa).

Therefore, the space complexity, in the worst case, is `O(m+n)`.

Additionally, it should be noted that the input grid is modified during the search process, which means the algorithm uses `O(1)` extra space aside from the stack space used by the recursion.