2510. Check if There is a Path With Equal Number of 0's And 1's

Medium | Array | Dynamic Programming | Matrix | Leetcode Link

Problem Description

The problem provides us with an `m x n` binary matrix, `grid`, indexed at `0`. We can navigate this grid, moving either right to the next column or down to the next row from our current position. The goal is to determine whether there exists a path from the top-left cell `(0, 0)` to the bottom-right cell `(m - 1, n - 1)` where the number of `0`s encountered is exactly equal to the number of `1`s encountered along the path.

Intuition

To solve this problem, we can employ a depth-first search (DFS) that keeps track of our position in the grid and the count of `1`s that we've seen so far. We define a criterion for a successful path: it has to reach the bottom-right corner of the grid and, upon reaching it, the count of `1`s must be equal to the count of `0`s. Since any path from `(0, 0)` to `(m - 1, n - 1)` is `m + n - 1` steps long and we want equal numbers of `1`s and `0`s, the total number of each must be half of `m + n - 1`. This is only possible if `m + n - 1` is even, hence we check for that early on and return `False` if it's not.

If the condition is met, we perform the DFS. The key to making the DFS manageable is to cache the results of previous paths using the `@cache` decorator to prevent re-computation and to cut off paths early if they become impossible. This happens if too many `1`s or `0`s are encountered before reaching the end, indicated by our counts exceeding half of `m + n - 1`.

The recursive DFS is defined as such:

- We increase our count of `1`s (`k`) as we encounter them along the path.
- If our current position is out of bounds, we return `False`.
- If the count of `1`s or `0`s exceeds half of the length of a potential correct path (`s`), we also return `False`.
- If we reach the bottom-right corner, we confirm if our count of `1`s is equal to `s` (as `0`s will automatically be equal due to path length constraints) and return `True` or `False` accordingly.
- At each cell, we explore both possible next cells (right and down), if either returns `True`, we have found a valid path.

The solution is initialized by setting `m` and `n` to the grid dimensions and computing `s`, which is half of the path length. DFS starts at `(0, 0)` with an initial count of `0`. If a path that satisfies the conditions is found, `dfs` will eventually return `True`, otherwise, `False`.

Solution Approach

The solution approach uses a recursive depth-first search (DFS) technique to navigate through the matrix. The DFS is augmented with memoization through the `@cache` decorator, which is a way to store the results of expensive function calls and return the cached result when the same inputs occur again, ensuring that each state is only computed once. This is critical for efficiency, especially in a matrix where there could be overlapping subproblems.

Here's the breakdown of the implementation:

- We initialize two variables `m` and `n` to store the number of rows and columns of the grid, respectively. Then we calculate `s` as half the length of a balanced path from `(0, 0)` to `(m - 1, n - 1)`, which is `(m + n - 1) / 2`. This division by 2 is represented by the right shift operator `s >>= 1`, which is an efficient way to divide by 2.
- We use the `dfs(i, j, k)` function where `i` and `j` are the current row and column in the grid, and `k` is the current count of `1`s encountered on the path.
- Inside the `dfs` function:
  - We first check if the current position is out of bounds (`i >= m or j >= n`), returning `False` since it is not a valid path.
  - We update the count of `1`s found (`k += grid[i][j]`).
  - If the number of `1`s (`k`) exceeds `s` or the number of `0`s (`i + j + 1 - k`) exceeds `s`, the path cannot be balanced, so we return `False`.
  - When the bottom-right corner (`i == m - 1 and j == n - 1`) is reached, we check if the number of `1`s is exactly `s`. If it is, we've found a valid path, otherwise, the path is invalid.
  - For all other cases, we recurse to the right `dfs(i, j + 1, k)` and down `dfs(i + 1, j, k)`. If either of these paths return `True`, we return `True`, indicating a valid path has been found.
- Before starting the DFS, we check if `s & 1` is truthy, meaning `m + n - 1` is odd and thus cannot be evenly split between `0`s and `1`s. In such a case, we return `False`.
- Finally, we call the `dfs(0, 0, 0)` function to start the path search from the top-left corner with `0` `1`s counted so far. If the function eventually returns `True`, then a balanced path exists, and we return `True`; otherwise, we return `False`.

The use of recursion and memoization (through `@cache`) is the key algorithmic pattern. This combination ensures that once a certain state (in terms of location and current count of `1`s) has been computed, it won't be recomputed unnecessarily, thus reducing the overall time complexity from exponential to polynomial.

Example Walkthrough

To illustrate the described solution approach, let's consider a small `3x3` binary matrix example:

```
grid = [
  [0, 1, 0],
  [0, 0, 1],
  [1, 0, 1]
]
```

- First, we determine `m` and `n`. In this case, `m = 3` and `n = 3`.
- We calculate `s` as half the length of a balanced path. Since the path from `(0, 0)` to `(m - 1, n - 1)` has `m + n - 1 = 5` steps, and `5` is odd, we can conclude right away that it's not possible to have a balanced path with an equal number of `0`s and `1`s. Therefore, in this situation, our algorithm would return `False`.

Suppose we had a scenario where `m + n - 1` is even. Let's change the grid to make that possible:

```
grid = [
  [0, 1],
  [1, 0]
]
```

- `m = 2`, `n = 2`, and `s = (2 + 2 - 1) / 2 = 1.5`. Since we deal with whole numbers, we can only have an equal number of `0`s and `1`s if `s` is an integer, so `3` would be rounded down to `1`.

Now, let's walk through a successful step-by-step DFS:

1. Start DFS with `dfs(0, 0, 0)`.
2. At `[0][0]`, our count `k` remains `0`. We call `dfs(0, 1, 0)` to move right and `dfs(1, 0, 0)` to move down.
3. The right move leads to `[0][1]`, which is a `1`, so `k` is incremented. The count `k` is now `1`, so we cannot move right anymore as it would leave the grid. We move down with `dfs(1, 1, 1)`.
4. The down move from step 2 leads to `[1][0]`, which is a `1`, so we increment `k`. We move right to `[1][1]` with `dfs(1, 1, 1)`.
5. Now at `[1][1]`, we are in the bottom-right corner. Here `k` is `1`, and since `s` is `1`, we have an equal number of `0`s and `1`s, which satisfies the conditions of the path.

Therefore, a valid path exists, and the function `dfs(0, 0, 0)` eventually returns `True`.

Python Solution

```python
from typing import List
from functools import lru_cache

class Solution:
    def is_there_a_path(self, grid: List[List[int]]) -> bool:
        # Calculate rows and columns for the provided grid
        rows, cols = len(grid), len(grid[0])
        # Compute the sum value for comparison during DFS
        target_sum = rows + cols - 1

        # Return False immediately if target_sum is odd, since we can't split into two equal integers
        if target_sum % 2 != 0:
            return False
        # The actual sum we need one of the paths to equal
        target_sum //= 2

        # Using memoization to avoid recomputing for the same cell
        @lru_cache(None)
        def dfs(row, col, path_sum):
            # If we are out of bounds, return False
            if row >= rows or col >= cols:
                return False

            # Increment path_sum by the value of the current cell
            path_sum += grid[row][col]

            # If the path_sum exceeds target_sum or if
            # the remaining cells aren't enough to complete the sum, return False
            if path_sum > target_sum or row + col + 1 - path_sum > target_sum:
                return False

            # If we've reached the bottom-right cell, check if the path sum equals the target sum
            if row == rows - 1 and col == cols - 1:
                return path_sum == target_sum

            # Recursively explore the path to the right and down
            return dfs(row + 1, col, path_sum) or dfs(row, col + 1, path_sum)

        # Initiate the recursive depth-first search from the top-left corner of the grid
        return dfs(0, 0, 0)

```

Java Solution

```java
public class Solution {
    // Variable 's' represents the target sum for the subsets.
    private int targetSum;
    // Variables 'rows' and 'cols' represent the dimensions of the grid.
    private int rows;
    private int cols;
    // The 'grid' stores the input grid.
    private int[][] grid;
    // The 'memo' stores previously computed results to avoid re-calculation during the DFS.
    private Boolean[][][] memo;

    // Method to determine if there is a path such that the sum of grid values is half the perimeter sum.
    public boolean isThereAPath(int[][] grid) {
        this.grid = grid;
        rows = grid.length;
        cols = grid[0].length;
        // Compute the sum of elements on the perimeter of the grid.
        targetSum = rows + cols - 1;
        // Initialize memoization array.
        memo = new Boolean[rows][cols][targetSum];
        // If the perimeter sum is odd, it's impossible to have two subsets with equal sum, return false.
        if (targetSum % 2 == 1) {
            return false;
        }
        // Halve the target sum since we want to find a subset with a sum equal to half the perimeter sum.
        targetSum >>= 1;
        // Start DFS search from the top-left corner of the grid.
        return dfs(0, 0, 0);
    }

    // Helper method to perform DFS on the grid.
    private boolean dfs(int i, int j, int currentSum) {
        // If the current cell is out of the grid bounds, return false.
        if (i >= rows || j >= cols) {
            return false;
        }
        // Add the value of the current cell to 'currentSum'.
        currentSum += grid[i][j];
        // If result has been computed before, return the stored value.
        if (memo[i][j][currentSum] != null) {
            return memo[i][j][currentSum];
        }
        // If the current sum exceeds half of the target sum or the complement exceeds it, return false.
        if (currentSum > targetSum || i + j + 1 - currentSum > targetSum) {
            return false;
        }
        // If we've reached the bottom-right corner, check if currentSum equals half of the target sum.
        if (i == rows - 1 && j == cols - 1) {
            return currentSum == targetSum;
        }
        // Perform DFS on the next element to the right and the bottom. Store result in 'memo' to avoid re-calculation.
        memo[i][j][currentSum] = dfs(i + 1, j, currentSum) || dfs(i, j + 1, currentSum);
        // Return the stored result.
        return memo[i][j][currentSum];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Check if there is a path in the grid where the sum equals the sum of leftover elements
    bool isThereAPath(vector<vector<int>>& grid) {
        int numRows = grid.size();    // Number of rows in the grid
        int numCols = grid[0].size(); // Number of columns in the grid
        int targetSum = numRows + numCols - 1; // Total possible sum for a path

        // Check if the targetSum is even, as we are looking for equal partition
        if (targetSum & 1) return false; // If the sum is odd, cannot be split equally

        targetSum >>= 1; // Divide the sum by 2 since we are looking for two equal halves

        // A 3D cache to store the states (result) of subproblems
        int cache[numRows][numCols][targetSum];
        memset(cache, -1, sizeof(cache)); // Initialize cache with -1

        // Define a recursive DFS function to explore the grid
        function<bool(int, int, int)> dfs = [&](int row, int col, int runningSum) -> bool {
            if (row >= numRows || col >= numCols) return false; // Out of grid bounds

            // Add the current grid value to the runningSum
            runningSum += grid[row][col];

            // Check if we already have a computed result for the current state in cache
            if (cache[row][col][runningSum] != -1) return cache[row][col][runningSum];

            // If the runningSum or the sum of the leftover elements exceeds the targetSum, return false
            if (runningSum > targetSum || row + col + 1 - runningSum > targetSum) return false;

            // Check if we reached the last cell and if the runningSum equals half the targetSum
            if (row == numRows - 1 && col == numCols - 1) return runningSum == targetSum;

            // Recur for the cells directly right and below the current cell
            cache[row][col][runningSum] = dfs(row + 1, col, runningSum) || dfs(row, col + 1, runningSum);
            // Save the result in cache

            return cache[row][col][runningSum]; // Return the cached result
        };

        // Start DFS from the top-left corner with an initial runningSum of 0
        return dfs(0, 0, 0);
    }
};

```

Typescript Solution

```typescript
// Define the type for a grid as a 2D array of numbers
type Grid = number[][];

// Define a function to check the existence of a path with sum equals to the sum of leftover elements
function isThereAPath(grid: Grid): boolean {
    const numRows: number = grid.length; // Number of rows in the grid
    const numCols: number = grid[0].length; // Number of columns in the grid
    let targetSum: number = numRows + numCols - 1; // Total possible sum for a path

    // Check if the targetSum is even, as we are looking for an equal partition
    if (targetSum % 2 !== 0) return false; // If the sum is odd, it cannot be split equally

    targetSum /= 2; // Divide the sum by 2 since we are looking for two equal halves

    // A 3D array cache to store the states (results) of subproblems
    const cache: number[][][] = Array.from({ length: numRows }, () =>
        Array.from({ length: numCols }, () => Array(targetSum + 1).fill(-1))
    );

    // Define a recursive DFS function to explore the grid
    const dfs = (row: number, col: number, runningSum: number): boolean => {
        if (row >= numRows || col >= numCols) return false; // Out of grid bounds

        // Add the current grid value to the runningSum
        runningSum += grid[row][col];

        // Check if we already have a computed result for the current state in cache
        if (cache[row][col][runningSum] !== -1) return cache[row][col][runningSum] === 1;

        // If the runningSum or the sum of the leftover elements exceeds the targetSum, return false
        if (runningSum > targetSum || row + col + 1 - runningSum > targetSum) return false;

        // Check if we reached the last cell and if the runningSum equals half the targetSum
        if (row === numRows - 1 && col === numCols - 1) return runningSum === targetSum;

        // Recur for the cells directly to the right and below the current cell
        const result = dfs(row + 1, col, runningSum) || dfs(row, col + 1, runningSum);
        cache[row][col][runningSum] = result ? 1 : 0; // Save the result in cache

        return result; // Return the result
    };

    // Start DFS from the top-left corner of the grid with an initial runningSum of 0
    return dfs(0, 0, 0);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the `isThereAPath` function is $O(mns)$, where `m` is the number of rows, `n` is the number of columns, and `s` is half the perimeter of the grid (sum of rows `m` and columns `n`, minus 1, all divided by 2).

This is because in the worst case, the `dfs` function is called for every possible position `(i, j)` and for every possible sum `k` up to `s`. Since there are `m*n` positions and up to `s` possible values for `k`, the result is `m*n*s`.

The recursive function might visit each cell multiple times with different values of `k`, but the memoization using `@cache` ensures that it will only recompute if it finds a cell with a distinct `k` value that it has not yet explored.

Space Complexity

The space complexity of the solution is $O(mns)$, which comes from the call stack used in recursion and the cache used for memoization.

The recursive depth of the stack can go as deep as `m + n` in the worst case (if a path involves all rows and columns). Additionally, the memoization employed by `@cache` will store results for each `(i, j, k)` tuple which has been computed and not yet been visited.

Since there are `m*n` positions and up to `s` distinct values for each position's sum `k`, the memoization cache can take up as much space as there are combinations of positions and sums, leading to a space complexity of $O(mns)$.