562. Longest Line of Consecutive One in Matrix

Medium Array Dynamic Programming Matrix Leetcode Link

Problem Description

The problem presents a binary matrix (`mat`), which is simply a grid made up of 0s and 1s. Our goal is to find the maximal length of a continuous line of 1s that could be arranged in any of four possible directions: horizontal, vertical, diagonal from top left to bottom right (referred to as 'diagonal'), and diagonal from top right to bottom left (referred to as 'anti-diagonal'). The task is to return the length of this longest line.

Intuition

To solve this problem, we can execute a comprehensive scan of the matrix. This involves checking each direction for each element in the matrix that is a 1 and updating counts of consecutive 1s in each direction.

The intuition for the solution can be broken down as follows:

1. For each cell that contains a 1 in the matrix, we have the potential to extend a line horizontally, vertically, diagonally, or anti-diagonally.

2. We can maintain four separate 2D arrays (of the same dimensions as the original matrix, with an extra buffer row and column to handle edge cases) to keep track of the maximum line length in each of the four directions at each cell.

   - `a[][]` for the vertical direction
   - `b[][]` for the horizontal direction
   - `c[][]` for the diagonal direction
   - `d[][]` for the anti-diagonal direction
3. By iterating over each cell in the matrix, we update the arrays for directions only if the current cell is a 1. For example, the length of the vertical line at `a[i][j]` is 1 plus the length at `a[i - 1][j]` if the cell at `mat[i - 1][j - 1]` is a 1.

4. Because we are only interested in consecutive 1s, whenever we hit a 0, the count resets to 0.

5. At every step, we update a variable `ans` to store the maximum line length found so far, resulting in having the maximum length by the time we end our scan.

6. The use of padding (an extra row and column) in arrays a, b, c, and d allows easy indexing without the need to check the matrix boundaries, which simplifies the logic and makes the code cleaner.

Following this approach allows us to resolve the problem with time complexity that is linear with respect to the number of cells in the matrix (O(m * n)) and with extra space for the four tracking matrices.

Solution Approach

The solution approach for this problem involves dynamic programming to keep track of the longest line of 1s for each of the four directions (horizontal, vertical, diagonal, and anti-diagonal) at every cell. Here is a breakdown of how the implementation works, tying back to the Python code provided:

1. **Initialization**: The solution first initializes four 2D arrays, `a`, `b`, `c`, and `d`, each with `m + 2` rows and `n + 2` columns. These arrays are used to store the maximum length of consecutive ones up to that point in the matrix for each direction. The extra rows and columns serve as a buffer that simplifies boundary condition handling.

2. **Iterative Processing**: The algorithm iterates through each cell in the matrix using two nested loops. The outer loop goes down each row, and the inner loop goes across each column. Only cells with a value of 1 are processed to extend the lines.

3. **Updating Directional Counts**: For each cell with a 1, the four arrays are updated as follows:

   - `a[i][j]` is updated to `a[i - 1][j] + 1`, incrementing the count of vertical consecutive ones from the top.
   - `b[i][j]` is updated to `b[i][j - 1] + 1`, incrementing the count of horizontal consecutive ones from the left.
   - `c[i][j]` is updated to `c[i - 1][j - 1] + 1`, incrementing the count of diagonal consecutive ones from the top left.
   - `d[i][j]` is updated to `d[i - 1][j + 1] + 1`, incrementing the count of anti-diagonal consecutive ones from the top right.
4. **Updating the Answer**: The maximum value of the four array cells at the current position (`a[i][j]`, `b[i][j]`, `c[i][j]`, `d[i][j]`) is compared with the current answer (`ans`). If any of them is greater, `ans` is updated. This ensures `ans` always holds the maximum line length found up to that point.

5. **Return the Result**: After completely scanning the matrix, the maximum length (`ans`) is returned as the answer.

The data structures used, in this case, are additional 2D arrays that help us track the solution state as we go, characteristic of dynamic programming. The implementation is relatively straightforward and relies on previous states to compute the current state. This ensures that at any given point in the matrix, we know the longest line of 1s that could be formed in any of the four directions without having to re-scan any part of the matrix, which is efficient and reduces the overall time complexity of the algorithm.

Example Walkthrough

Let's illustrate the solution approach with a small binary matrix example:

```
mat = [
    [0, 1, 1, 0],
    [0, 1, 1, 1],
    [1, 0, 0, 1]
]
```

Following the steps outlined in the solution approach:

1. **Initialization**: We create four extra 2D arrays `a`, `b`, `c`, and `d`, each with dimensions `4 x 5` to accommodate the buffer rows and columns.

2. **Iterative Processing**: We step through each cell in the matrix one by one.

3. **Updating Directional Counts and Answer**: We update the counts in our four arrays only at positions where the corresponding cell in `mat` is `1`. Assume `ans = 0` at the start.

To better understand, let's look at the updates after processing the first two rows:

After the first row:

```
a (vertical)       b (horizontal)    c (diagonal)      d (anti-diagonal)
0 0 0 0 0          0 0 0 0 0         0 0 0 0 0          0 0 0 0 0
0 0 1 2 0          0 0 0 1 2         0 0 0 0 1          0 0 1 2 0
0 0 0 0 0          0 0 0 0 0         0 0 0 0 0          0 0 0 0 0
ans = 2
```

Here, `ans` is 2, as that is the largest count in any of the arrays.

After the second row:

```
a (vertical)       b (horizontal)    c (diagonal)      d (anti-diagonal)
0 0 0 0 0          0 0 0 0 0         0 0 0 0 0          0 0 0 0 0
0 0 1 2 0          0 0 0 1 2         0 0 0 0 1          0 0 1 2 0
0 0 2 3 0          0 0 0 1 2 3       0 0 0 1 2          0 0 0 3 1
ans = 3
```

After processing the cell `mat[1][3]`, `ans` updates to 3, as that is now the highest value found.

4. **Return the Result**: After completely iterating over the matrix, we find that the `ans` is `3`, which indicates the length of the longest continuous line of 1s. We would return `3` in this case.

And so, by iterating over the matrix and updating our directional counts, we can determine that the longest continuous line of `1`s in `mat` spans 3 cells and is found horizontally in the second row.

Python Solution

```python
from typing import List  # Import typing to use List type hint

class Solution:
    def longestLine(self, matrix: List[List[int]]) -> int:
        # Get the dimensions of the matrix
        rows, cols = len(matrix), len(matrix[0])

        # Initialize four matrices to keep track of continuous 1s in all four directions:
        # horizontal, vertical, diagonal, anti-diagonal
        horizontal = [[0] * (cols + 2) for _ in range(rows + 2)]
        vertical = [[0] * (cols + 2) for _ in range(rows + 2)]
        diagonal = [[0] * (cols + 2) for _ in range(rows + 2)]
        anti_diagonal = [[0] * (cols + 2) for _ in range(rows + 2)]

        # Variable to store the maximum length of continuous 1s
        max_length = 0

        # Iterate through the matrix
        for i in range(1, rows + 1):
            for j in range(1, cols + 1):
                # Value of the current cell in the input matrix
                value = matrix[i - 1][j - 1]
                if value == 1:
                    # Update counts for all four directions by adding 1 to the counts from previous
                    # relevant cells (up, left, top-left diagonal, top-right diagonal).
                    horizontal[i][j] = horizontal[i][j - 1] + 1
                    vertical[i][j] = vertical[i - 1][j] + 1
                    diagonal[i][j] = diagonal[i - 1][j - 1] + 1
                    anti_diagonal[i][j] = anti_diagonal[i - 1][j + 1] + 1

                    # Update the max_length for the current cell's longest line of continuous 1s.
                    max_length = max(max_length, horizontal[i][j], vertical[i][j],
                                     diagonal[i][j], anti_diagonal[i][j])

        # Return the maximum length of continuous 1s found.
        return max_length

```

Java Solution

```java
class Solution {
    public int longestLine(int[][] mat) {
        // Get the number of rows and columns in the matrix
        int rows = mat.length, cols = mat[0].length;
        int[][] horizontal = new int[rows + 2][cols + 2];
        int[][] vertical = new int[rows + 2][cols + 2];
        int[][] diagonal = new int[rows + 2][cols + 2];
        int[][] antiDiagonal = new int[rows + 2][cols + 2];

        // Initialize a variable to keep track of the maximum length
        int maxLength = 0;

        // Iterate over each cell in the matrix
        for (int i = 1; i <= rows; ++i) {
            for (int j = 1; j <= cols; ++j) {
                // If the current cell has a value of 1
                if (mat[i - 1][j - 1] == 1) {
                    // Update the counts for each direction (horizontal, vertical, diagonal, antiDiagonal)
                    horizontal[i][j] = horizontal[i][j - 1] + 1;      // Left
                    vertical[i][j] = vertical[i - 1][j] + 1;          // Up
                    diagonal[i][j] = diagonal[i - 1][j - 1] + 1;      // Top-left
                    antiDiagonal[i][j] = antiDiagonal[i - 1][j + 1] + 1;  // Top-right

                    // Update the maximum length if a higher count is found
                    maxLength = getMax(maxLength, horizontal[i][j], vertical[i][j], diagonal[i][j], antiDiagonal[i][j]);
                }
            }
        }
        // Return the maximum length of a line of consecutive ones
        return maxLength;
    }

    // Helper function to calculate the maximum value
    private int getMax(int... values) {
        int max = 0;
        for (int value : values) {
            max = Math.max(max, value);
        }
        return max;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // For max()
using namespace std;

class Solution {
public:
    int longestLine(vector<vector<int>>& matrix) {
        int rows = matrix.size(), cols = matrix[0].size();

        // Create 2D vectors with extra padding to handle indices during DP calculations
        vector<vector<int>> vertical(rows + 2, vector<int>(cols + 2, 0));
        vector<vector<int>> horizontal(rows + 2, vector<int>(cols + 2, 0));
        vector<vector<int>> diagonal(rows + 2, vector<int>(cols + 2, 0));
        vector<vector<int>> antiDiagonal(rows + 2, vector<int>(cols + 2, 0));

        int maxLength = 0; // To keep track of the longest line of consecutive ones

        // Iterate through each cell of the matrix to fill DP tables
        for (int i = 1; i <= rows; ++i) {
            for (int j = 1; j <= cols; ++j) {
                // Only process the cell if it contains a '1'
                if (matrix[i - 1][j - 1] == 1) {
                    // Update the dynamic programming tables
                    // Compute number of consecutive ones in all directions
                    vertical[i][j] = vertical[i - 1][j] + 1; // Count consecutive ones vertically
                    horizontal[i][j] = horizontal[i][j - 1] + 1; // Count consecutive ones horizontally
                    diagonal[i][j] = diagonal[i - 1][j - 1] + 1; // Count consecutive ones diagonally
                    antiDiagonal[i][j] = antiDiagonal[i - 1][j + 1] + 1; // Count consecutive ones anti-diagonally

                    // Check if the current count is larger than the current maximum length
                    int currentMax = max(vertical[i][j], max(horizontal[i][j],
                                         max(diagonal[i][j], antiDiagonal[i][j])));
                    maxLength = max(maxLength, currentMax);
                }
            }
        }

        // Return the length of the longest consecutive line of ones found
        return maxLength;
    }
};

```

Typescript Solution

```typescript
function longestLine(matrix: number[][]): number {
    let rows = matrix.length;
    let cols = matrix[0].length;

    // Initialize 2D arrays with extra padding to handle indices during dynamic programming calculations
    let vertical = Array.from({ length: rows + 2 }, () => Array(cols + 2).fill(0));
    let horizontal = Array.from({ length: rows + 2 }, () => Array(cols + 2).fill(0));
    let diagonal = Array.from({ length: rows + 2 }, () => Array(cols + 2).fill(0));
    let antiDiagonal = Array.from({ length: rows + 2 }, () => Array(cols + 2).fill(0));

    let maxLength = 0; // To keep track of the longest line of consecutive ones

    // Iterate through each cell of the matrix to fill the DP arrays
    for (let i = 1; i <= rows; i++) {
        for (let j = 1; j <= cols; j++) {
            // Only process the cell if it contains a '1'
            if (matrix[i - 1][j - 1] === 1) {
                // Update the dynamic programming arrays
                // Compute the number of consecutive ones in all directions
                vertical[i][j] = vertical[i - 1][j] + 1; // Count consecutive ones vertically
                horizontal[i][j] = horizontal[i][j - 1] + 1; // Count consecutive ones horizontally
                diagonal[i][j] = diagonal[i - 1][j - 1] + 1; // Count consecutive ones diagonally
                antiDiagonal[i][j] = antiDiagonal[i - 1][j + 1] + 1; // Count consecutive ones anti-diagonally

                // Check if the current count is larger than the maximum length found so far
                let currentMax = Math.max(vertical[i][j], Math.max(horizontal[i][j],
                                 Math.max(diagonal[i][j], antiDiagonal[i][j])));
                maxLength = Math.max(maxLength, currentMax);
            }
        }
    }

    // Return the length of the longest consecutive line of ones found
    return maxLength;
}

```

Time and Space Complexity

The time complexity of the provided code is `O(m * n)`, where `m` is the number of rows and `n` is the number of columns in the input matrix `mat`. This complexity arises because the code iterates over each cell of the matrix exactly once, performing a constant number of operations for each cell.

The space complexity of the code is also `O(m * n)` because it creates four auxiliary matrices (`a`, `b`, `c`, `d`), each of the same size as the input matrix `mat`. These matrices are used to keep track of the length of consecutive ones in four directions - horizontal, vertical, diagonal, and anti-diagonal.