2371. Minimize Maximum Value in a Grid

Hard | Union Find | Graph | Topological Sort | Array | Matrix | Sorting | Leetcode Link

Problem Description

The problem presents a challenge to minimize the maximum value in a given `m x n` matrix, `grid`, while maintaining the relative ordering of numbers in their respective rows and columns. In this context, "relative order" means that if a number is greater than another number in the same row or column in the original matrix, it must remain greater in the transformed matrix. The objective is to do so by replacing each element in the matrix with a new positive integer and ensuring that the new maximum number in the matrix is as small as possible.

A fundamental requirement is that the original matrix contains distinct positive integers. This constraint simplifies the problem, as we do not need to consider the case where two elements are equal. The challenge is to determine the smallest possible replacements that satisfy the aforementioned relative order conditions.

Intuition

To maintain the relative order within rows and columns, we look at the individual elements' ranks when compared to other elements within the same rows and columns. The key observation is that the smallest number can be replaced by 1, the second smallest by 2, and so on, without changing their relative positions. However, simply increasing each subsequent element by 1 does not always work, especially when an element is not the smallest in its row or column.

The approach for finding the solution is twofold: Firstly, sort all of the elements in the grid based on their value while keeping track of their original positions. This allows us to process the elements in increasing order, ensuring that each replacement maintains the overall relative ordering structure of the grid.

Secondly, maintain two arrays `row_max` and `col_max`, where `row_max[i]` indicates the largest number assigned so far in row `i`, and `col_max[j]` the largest number in column `j`. As we process each element in sorted order, we determine its new value by taking the maximum of `row_max[i]` and `col_max[j]` for its position `(i, j)` and then add 1 to it. This ensures that the new number will be larger than any previously assigned number in the same row or column, maintaining the relative order. The two arrays are then updated with this new value to reflect the new maximum values for that row and column.

By using this method, we can iterate through all the elements in the grid only once, after sorting, and consistently assign the minimum required number that maintains both the ordering and the constraint of minimizing the maximum number in the resultant matrix.

Solution Approach

The solution approach takes advantage of a few powerful algorithms and data structures to organize and process data efficiently. Here's how the solution implements the approach step by step:

1. **Sorting**: The original matrix elements are expanded into a list of tuples `nums`, where each tuple contains an element value along with its row and column indices. This list is then sorted primarily by the value, which means the tuples will be arranged in ascending order according to the elements of the matrix they represent.

```
nums = [(v, i, j) for i, row in enumerate(grid) for j, v in enumerate(row)]
nums.sort()
```

   This organizing strategy is crucial because it allows for the iteration over the matrix elements in order of their size. Since each element is associated with its original position, we can apply the exact replacements required, maintaining the relative order.

2. **Tracking Row and Column Maximums**: Two arrays `row_max` and `col_max` are used to keep track of the highest value that has been assigned to each row and column, respectively. Initiated with zeros, these arrays will update with each assignment made in the following steps:

```
row_max = [0] * m
col_max = [0] * n
```

3. **Constructing the Result Matrix**: A new matrix `ans` is initialized with zero values, where the solution will be built incrementally.

```
ans = [[0] * n for _ in range(m)]
```

4. **Iterating and Assigning New Values**: The algorithm then iterates over this sorted list of tuples. For each tuple, which corresponds to an element from the original grid, we determine the replacement number. This number is one greater than the maximum of the current values of `row_max[i]` and `col_max[j]`. We then update both `row_max` and `col_max` at index `i` and `j` to reflect this new maximum.

```
for _, i, j in nums:
    ans[i][j] = max(row_max[i], col_max[j]) + 1
    row_max[i] = col_max[j] = ans[i][j]
```

In this implementation, Python list comprehensions, tuple unpacking, and sorting mechanisms are powerful tools used to streamline the solution process. This approach ensures that each element's relative value to its neighbors is preserved while the overall maximum value in the matrix is minimized. The end result is a matrix `ans` that meets all the stipulated requirements and can be returned as the solution to the problem.

Thus, the solution makes effective use of familiar data structures (such as lists) and algorithmic patterns (like sorting and iteration) to solve a relatively complex problem in an efficient and straightforward manner.

Example Walkthrough

Let's take a `2x3` matrix as an example to illustrate the solution approach.

Imagine we have the following matrix `grid`:

```
8 4 6
3 5 9
```

The challenge is to minimize the maximum number after replacing numbers while maintaining their relative ordering.

1. **Sorting**:

   - Expand the elements of `grid` into a list of tuples containing the value with its row and column index:

```
nums = [(8, 0, 0), (4, 0, 1), (6, 0, 2), (3, 1, 0), (5, 1, 1), (9, 1, 2)]
```

   - After sorting `nums` by value, we get:

```
nums = [(3, 1, 0), (4, 0, 1), (5, 1, 1), (6, 0, 2), (8, 0, 0), (9, 1, 2)]
```

2. **Tracking Row and Column Maximums**:

   - Initialize `row_max` and `col_max` to keep track of the max values so far:

```
row_max = [0, 0]
col_max = [0, 0, 0]
```

3. **Constructing the Result Matrix**:

   - Initialize the `ans` matrix with zeroes to build the solution:

```
ans = [[0, 0, 0], [0, 0, 0]]
```

4. **Iterating and Assigning New Values**:

   - Iterate over sorted `nums` and replace values in `ans` while updating `row_max` and `col_max`:
     - For `(3, 1, 0)` → `ans` becomes:

```
ans = [[0, 0, 0], [1, 0, 0]]
```

       And `row_max` and `col_max` update to:

```
row_max = [0, 1]
col_max = [1, 0, 0]
```

     - For `(4, 0, 1)` → `ans` becomes:

```
ans = [[0, 2, 0], [1, 0, 0]]
```

       And `row_max` and `col_max` update to:

```
row_max = [2, 1]
col_max = [1, 2, 0]
```

     - Repeat this process for each element in `nums`.

The final `ans` matrix after completing the iterations will look like this:

```
2 2 3
1 2 4
```

In this new `ans` matrix, the relative ordering is maintained, and the maximum value, which is `4`, is minimized compared to the original `grid`. The `ans` matrix can be returned as the solution to the problem.

Python Solution

```python
class Solution:
    def min_score(self, grid):
        """
        This method finds the minimum score for each cell in the grid such that
        each row and column contains unique scores.
        It sorts the cells initially and uses those values to ensure the uniqueness.
        """
        # Get the dimensions of the grid
        num_rows, num_cols = len(grid), len(grid[0])

        # Flatten the grid into a list of tuples (value, row_index, col_index)
        flatten_grid = [(value, row_index, col_index) for row_index, row in enumerate(grid) for col_index, value in enumerate(row)]

        # Sort the flatten_grid based on the cell values
        flatten_grid.sort()

        # Initialize lists to keep track of the maximum score in each row and column
        row_max = [0] * num_rows
        col_max = [0] * num_cols

        # Create an answer grid initialized with zeros
        ans_grid = [[0] * num_cols for _ in range(num_rows)]

        # Iterate over sorted cells and assign the minimum required score
        for value, row_index, col_index in flatten_grid:
            # The score for the cell is 1 more than the max of the row and column seen so far
            ans_grid[row_index][col_index] = max(row_max[row_index], col_max[col_index]) + 1
            # Update the current row and column max values
            row_max[row_index] = col_max[col_index] = ans_grid[row_index][col_index]

        # Return the answer grid with all scores filled in
        return ans_grid

# Example usage:
# solution_instance = Solution()
# grid = [[3, 1], [2, 4]]
# print(solution_instance.min_score(grid))  # Should output the minimum score grid based on the input grid
```

Java Solution

```java
class Solution {
    public int[][] minScore(int[][] grid) {
        int rows = grid.length; // Number of rows in the grid
        int cols = grid[0].length; // Number of columns in the grid

        // This list will hold the value, row, and column for each cell in the grid
        List<int[]> cellDetails = new ArrayList<>();

        // Populate the list with cell details
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                cellDetails.add(new int[] {grid[i][j], i, j});
            }
        }

        // Sort the cell details based on the cell value
        Collections.sort(cellDetails, (a, b) -> a[0] - b[0]);

        // Arrays to keep track of the maximum score in each row and column
        int[] rowMaxScores = new int[rows];
        int[] colMaxScores = new int[cols];

        // Answer grid to hold the minimum required scores
        int[][] minScores = new int[rows][cols];

        // Update the answer grid with the minimum required scores,
        // looping through the cells by increasing order of their values
        for (int[] cell : cellDetails) {
            int cellValue = cell[0]; // The value of the cell
            int row = cell[1]; // The row index of the cell
            int col = cell[2]; // The column index of the cell

            // Compute the minimum score for this cell based on the max scores in the current row and column
            minScores[row][col] = Math.max(rowMaxScores[row], colMaxScores[col]) + 1;

            // Update the row and column max scores with the newly computed score
            rowMaxScores[row] = minScores[row][col];
            colMaxScores[col] = minScores[row][col];
        }

        // Return the completed grid with minimum required scores
        return minScores;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    vector<vector<int>> minScore(vector<vector<int>>& grid) {
        // Create a vector that will store the value and its coordinates
        vector<tuple<int, int, int>> cells_with_values;
        int rows = grid.size(), cols = grid[0].size();

        // Populate the vector with the grid values and their corresponding coordinates
        for (int row = 0; row < rows; ++row) {
            for (int col = 0; col < cols; ++col) {
                cells_with_values.push_back({grid[row][col], row, col});
            }
        }

        // Sort the vector of tuples based on the cell values in non-decreasing order
        sort(cells_with_values.begin(), cells_with_values.end());

        // Create vectors to keep track of the maximum scores for each row and column
        vector<int> row_max_scores(rows);
        vector<int> col_max_scores(cols);

        // Initialize the answer grid with the same dimensions as the input grid
        vector<vector<int>> answer(rows, vector<int>(cols));

        // Iterate over the sorted cell values
        for (auto [value, row, col] : cells_with_values) {
            // Calculate the score for the current cell, which is 1 plus the max score of the current row or column
            answer[row][col] = max(row_max_scores[row], col_max_scores[col]) + 1;

            // Update the max score for the current row and column to be the score of the current cell
            row_max_scores[row] = col_max_scores[col] = answer[row][col];
        }

        // Return the populated answer grid
        return answer;
    }
};
```

Typescript Solution

```typescript
function minScore(grid: number[][]): number[][] {
    // Get the dimensions of the grid
    const rows = grid.length;
    const cols = grid[0].length;

    // Create an auxiliary array to hold grid elements and their coordinates
    const elements = [];
    for (let i = 0; i < rows; ++i) {
        for (let j = 0; j < cols; ++j) {
            elements.push({
                value: grid[i][j],
                row: i,
                col: j
            });
        }
    }

    // Sort the elements array based on the value in ascending order
    elements.sort((a, b) => a.value - b.value);

    // Create arrays to keep track of the maximum score in each row and column
    const rowMaxScores = new Array(rows).fill(0);
    const colMaxScores = new Array(cols).fill(0);

    // Initialize an answer grid with the same dimensions as the input grid
    const answerGrid = Array.from({ length: rows }, () => new Array(cols));

    // Process each element in the sorted list
    for (const element of elements) {
        // Calculate the score for the current element's position
        const score = Math.max(rowMaxScores[element.row], colMaxScores[element.col]) + 1;

        // Update the answer grid with the new score
        answerGrid[element.row][element.col] = score;

        // Update the maximum score counters for the current row and column
        rowMaxScores[element.row] = score;
        colMaxScores[element.col] = score;
    }

    // Return the completed answer grid
    return answerGrid;
}
```

Time and Space Complexity

Time Complexity

The given code has several distinct operations, each contributing to the overall time complexity.

1. First, the list comprehension iterates over each element in the $m \times n$ grid to create a list of tuples. This operation has a time complexity of $O(m * n)$.

2. Next, the `sort` operation on this list of tuples has a time complexity of $O(m * n \log(m * n))$, since it sorts the list with respect to the values in the grid.

3. Then, the code iterates over the sorted list `nums` with $m * n$ elements, performing constant-time operations inside the loop. This contributes $O(m * n)$ to the time complexity.

4. The updates to `row_max` and `col_max` are also constant-time operations, happening $m * n$ times.

Combining all the above steps, the overall time complexity of the code is $O(m * n \log(m * n)) + O(m * n) + O(m * n)$, which simplifies to $O(m * n \log(m * n))$ since the log term will dominate at scale.

Space Complexity

The space complexity can be analyzed based on the data structures used:

1. The list `nums` which stores the tuples has a space complexity of $O(m * n)$ because it stores each element of the grid.

2. Two arrays, `row_max` and `col_max`, each with a length of m and n respectively, resulting in $O(m)$ and $O(n)$ space used.

3. The `ans` list is a 2D list of the same size as the input grid, yielding a space complexity of $O(m * n)$.

Therefore, the total space complexity is $O(m * n) + O(m) + O(n)$, which simplifies to $O(m * n)$ when m and n are of the same order.