2711. Difference of Number of Distinct Values on Diagonals

Medium Array Hash Table Matrix

Leetcode Link

Problem Description

The task is to compute a matrix `answer` based on another 2D `grid` given as input. The `grid` is a two-dimensional array with `n` rows and `c` columns, and we want to create a new 2D array `answer` of the same dimensions.

For each cell in the `answer` matrix at position `(r, c)`, the value is determined as follows:

- `topLeftFr[r][c]`: count the number of distinct values that appear in the cells on the top-left diagonal of cell `(r, c)` in the original `grid`. This diagonal includes cells `(x, y)` from the top row or the leftmost column cutting through to just above `(r, c)` such that `x < r and y < c`.
- `bottomRightFr[r][c]`: count the number of distinct values that appear in the cells on the bottom-right diagonal of cell `(r, c)` in the original `grid`. This diagonal includes cells `(x, y)` starting just below `(r, c)` running down to the bottom right corner of the grid such that `x > r and y > c`.

The value in `answer[r][c]` is the absolute difference between these two counts: `|topLeftFr[r][c] - bottomRightFr[r][c]|`.

The problem requires returning the matrix `answer`, which is constructed by performing these computations for every cell `(r, c)` of `grid`.

Intuition

The intuition behind the solution is to focus on each cell `(r, c)` individually and to accumulate the distinct values along its top-left and bottom-right diagonals. Using sets is helpful because sets naturally maintain unique elements, thereby making the counting of distinct values straightforward.

For the top-left diagonal, we start from the current cell `(r, c)` and move upwards and to the left (`x -= 1, y -= 1`) until we either reach the first row or the first column.

For the bottom-right diagonal, we start from the current cell `(r, c)` and move downwards and to the right (`x += 1, y += 1`) until we either reach the last row or the last column.

At each step for both diagonals, we add the values of `grid[x][y]` to their respective sets. Once all unique values are accumulated, the sizes of the sets represent the count of distinct values in each diagonal. We calculate the absolute difference of these counts and assign it to `answer[r][c]`.

This process is repeated for every cell in the `grid`, which eventually yields the completed `answer` matrix. The brute force nature of this solution is acceptable given the problem constraints, and it directly translates the problem's requirements into algorithmic steps to find the solution.

Solution Approach

The solution as implemented above employs a straightforward, brute-force approach. To compute the `answer[r][c]`, the algorithm inspects two distinct diagonals for each cell `(r, c)` in the `grid`. Here is a step-by-step breakdown of the approach:

1. Initialize a matrix `ans` with the same dimensions as `grid` but filled with zeros. This will store the final results.
2. Traverse every cell `(r, c)` in the grid. Use nested loops, the outer loop running through the rows, and the inner loop going through the columns.
3. To find `topLeftFr[r][c]`, initialize an empty set `ts`. Then, iterate from the current cell upwards and leftward diagonally (decreasing both row and column indices). For each cell encountered, add the value of the cell from `grid[x][y]` to the set `ts`. Loop stops when reaching the top row or the leftmost column. After iteration, `tl` is set as the length of the set `ts`, which is the count of unique values in the top-left diagonal.

```
1  ts, x, y = set(), r - 1, c - 1
2  while x >= 0 and y >= 0:
3      ts.add(grid[x][y])
4      x, y = x - 1, y - 1
5  tl = len(ts)
```

4. Repeat a similar process to find `bottomRightFr[r][c]`. Initialize another empty set `s`, and iterate from the current cell downwards and rightward diagonally (increasing both row and column indices). Add the value from `grid[x][y]` to set `s` until reaching the bottom row or the rightmost column. Assign `br` as the length of this set.

```
1  s, x, y = set(), r + 1, c + 1
2  while x < l and y < w:
3      s.add(grid[x][y])
4      x, y = x + 1, y + 1
5  br = len(s)
```

5. Calculate the absolute difference between `tl` and `br` and assign it to `ans[i][j]`. This step executes for every cell `(r, c)`, filling the `ans` matrix.

```
1  ans[i][j] = abs(tl - br)
```

6. After filling in all values, return the `ans` matrix.

Using this method, every cell in the `grid` is visited, and its corresponding diagonals are processed to determine the values of `answer[r][c]`. The utilization of sets for counting unique elements is pivotal because it eliminates the need for manual checks for duplicates, simplifying the code and ensuring accuracy.

Example Walkthrough

Let's go through a small example to illustrate the solution approach.

Suppose we have the following `grid` matrix:

```
1  grid =
2  [ [1, 2, 3],
3    [4, 5, 6],
4    [7, 8, 1] ]
```

We want to construct an `answer` matrix where each element is the absolute difference between the count of distinct numbers in its top-left and bottom-right diagonals.

Step-by-step Process:

1. Initialize an `answer` matrix with zeros having the same dimensions as `grid`:

```
1  answer = [
2    [0, 0, 0],
3    [0, 0, 0],
4    [0, 0, 0]
5  ]
```

2. Start iterating over each cell `(r, c)` of the grid. First with cell `(0, 0)`:
   - There is no top-left diagonal, so `topLeftFr = 0`.
   - For the bottom-right diagonal, only `1` is on the diagonal, so `bottomRightFr = 1`.
   - `answer[0][0] = |0 - 1| = 1`.
   - Update the `answer` matrix:

```
1  answer = [
2    [1, 0, 0],
3    [0, 0, 0],
4    [0, 0, 0]
5  ]
```

3. Move to cell `(0, 1)`:
   - Again, no top-left diagonal, so `topLeftFr = 0`.
   - The bottom-right diagonal has `1` and `5`, so `bottomRightFr = 2`.
   - `answer[0][1] = |0 - 2| = 2`
   - Update the `answer` matrix:

```
1  answer = [
2    [1, 2, 0],
3    [0, 0, 0],
4    [0, 0, 0]
5  ]
```

Now we'll do the same for cell `(1,1)`, which has both top-left and bottom-right diagonals:

4. Inspect the cell `(1, 1)`:
   - The top-left diagonal contains `4` and `1` (so `topLeftFr[1][c] = 2`).
   - The bottom-right diagonal contains only `1`, hence `bottomRightFr[1][c] = 1`.
   - `answer[1][1] = |2 - 1| = 1`.

Update the `answer` matrix:

```
1  answer = [
2    [1, 2, 0],
3    [0, 1, 0],
4    [0, 0, 0]
5  ]
```

5. Continue the process for each cell, calculating the unique counts on both diagonals and computing their absolute difference.

After completing this process for all cells, we would obtain the final `answer` matrix:

```
1  answer = [
2    [1, 2, 1],
3    [1, 1, 0],
4    [1, 0, 0]
5  ]
```

Each step of the process follows the explanation in the solution approach, utilizing the property of sets to count unique elements along the top-left and bottom-right diagonals for each cell and calculating the absolute difference between these unique counts for the `answer` matrix.

Python Solution

```
1  class Solution:
2      def differenceOfDistinctValues(self, grid: List[List[int]]) -> List[List[int]]:
3          # Get the grid dimensions
4          rows, cols = len(grid), len(grid[0])
5
6          # Initialize the answer grid with zeros, matching the input grid dimensions.
7          answer_grid = [[0 * cols for _ in range(rows)]
8
9          # Iterate over each cell in the grid.
10         for row_in in range(rows):
11             for col_in in range(cols):
12                 # Sets to hold unique values for traversing diagonally up-left.
13                 up_left = set()
14                 unique_values_up_left = set()  # To store unique values seen so far up-left.
15
16                 # Traverse diagonally to the top-left corner, collecting unique values.
17                 while x_up >= 0 and y_up >= 0:
18                     x_up, y_up = x_up - 1, y_up - 1
19                     unique_values_up_left.add(grid[x_up][y_up])
20                     up_left.add(grid[x_up][y_up])
21
22                 # Count the unique values in the top-left diagonal.
23                 count_up_left = len(unique_values_up_left)
24
25                 # Reset the variables for traversing diagonally down-right.
26                 x_down, y_down = row_in, col_in
27                 unique_values_down_right = set()  # To store unique values seen so far down-right.
28
29                 # Traverse diagonally to the bottom-right corner, collecting unique values.
30                 while x_down < rows and y_down < cols:
31                     x_down, y_down = x_down + 1, y_down + 1
32                     unique_values_down_right.add(grid[x_down][y_down])
33
34                 # Count the unique values in the bottom-right diagonal.
35                 count_down_right = len(unique_values_down_right)
36
37                 # Calculate the absolute difference and update the value in the answer grid.
38                 answer_grid[row_in][col_in] = abs(count_up_left - count_down_right)
39
40         # Return the answer grid populated with differences of distinct values.
41         return answer_grid
42
43 # Notes:
44 # 1. The variable names have been improved for readability.
45 # 2. Added comments to explain each section of code.
46 # 3. The code logic and method names are unchanged since the request was only to rewrite the code with better variable naming and a
47 # 4. Used zero-indexed while loops for traversing the diagonals to correctly access the grid indices.
```

Java Solution

```
1  class Solution {
2      public int[][] differenceOfDistinctValues(int[][] grid) {
3          // Get the number of rows and columns in the grid.
4          int rows = grid.length, cols = grid[0].length;
5          // Initialize the answer grid with the same dimensions.
6          int[][] answerGrid = new int[rows][cols];
7
8          // Iterate over each cell in the grid.
9          for (int r = 0; r < rows; ++r) {
10             for (int c = 0; c < cols; ++c) {
11                 // Compute the distinct count in the top-left diagonal direction.
12                 int topLeftDistinctCount = calculateDistinctCount(grid, r, c, -1, -1);
13                 // Compute the distinct count in the bottom-right diagonal direction.
14                 int bottomRightDistinctCount = calculateDistinctCount(grid, r, c, 1, 1);
15                 // Store the absolute difference of the distinct counts in both diagonal directions.
16                 answerGrid[r][c] = Math.abs(topLeftDistinctCount - bottomRightDistinctCount);
17             }
18         }
19
20         // Return the filled answer grid.
21         return answerGrid;
22     }
23
24     // Helper method to calculate the distinct count in a diagonal direction.
25     private int calculateDistinctCount(int[][] grid, int row, int col, int direction) {
26         // Create a set to keep track of the distinct values.
27         Set<Integer> distinctValues = new HashSet<>();
28         // Continue moving in the diagonal direction until the grid boundaries are reached.
29         while (row >= 0 && col >= 0 && row < grid.length && col < grid[0].length) {
30             // Add the current value to the set of distinct values.
31             distinctValues.add(grid[row][col]);
32             // Move in the diagonal direction specified by 'direction'.
33             row += direction;
34             col += direction;
35         }
36         // Return the size of the set, i.e., the number of distinct values.
37         return distinctValues.size();
38     }
39 }
```

C++ Solution

```
1  class Solution {
2  public:
3      vector<vector<int>> differenceOfDistinctValues(vector<vector<int>>& grid) {
4          int rows = grid.size();  // Number of rows in the input grid
5          int cols = grid[0].size();  // Number of columns in the input grid
6          vector<vector<int>> answer(rows, vector<int>(cols));
7
8          // Iterate over every cell in the grid
9          for (int i = 0; i < rows; ++i) {
10             for (int j = 0; j < cols; ++j) {
11                 unordered_set<int> topLeftValues;  // Set to store distinct values in the top-left diagonal
12                 unordered_set<int> bottomRightValues;  // Set to store distinct values in the bottom-right diagonal
13
14                 // Traverse the top-left diagonal from the current cell
15                 for (int x = i - 1, y = j - 1; x >= 0 && y >= 0; --x, --y) {
16                     topLeftValues.insert(grid[x][y]);
17                 }
18
19                 // Count distinct values in the top-left diagonal from the current cell
20                 int topLeftCount = topLeftValues.size();
21
22                 // Traverse the bottom-right diagonal from the current cell
23                 for (int x = i + 1, y = j + 1; x < rows && y < cols; ++x, ++y) {
24                     bottomRightValues.insert(grid[x][y]);
25                 }
26
27                 // Count distinct values in the bottom-right diagonal from the current cell
28                 int bottomRightCount = bottomRightValues.size();
29
30                 // Store the absolute difference of the counts in the answer grid
31                 answer[i][j] = abs(topLeftCount - bottomRightCount);
32             }
33         }
34         return answer;  // Return the final grid containing absolute differences
35     }
36 };
37
```

Typescript Solution

```
1  function differenceOfDistinctValues(grid: number[][]): number[][] {
2      // Get the dimensions of the grid
3      const rows = grid.length;
4      const columns = grid[0].length;
5
6      // Initialize an answer grid of the same dimensions with zeroes
7      const answerGrid: number[][] = Array.from({ length: rows }, () => Array(columns).fill(0));
8
9      // Iterate over each cell of the grid
10     for (let i = 0; i < rows; ++i) {
11         for (let j = 0; j < columns; ++j) {
12             // Initialize row and column index for top-left diagonal
13             let [topLeftRow, topLeftCol] = [i - 1, j - 1];
14             let topLeftSet = new Set();
15
16             // Create a set to store unique values in the top-left diagonal
17             const topLeftUnique = new Set<number>();
18
19             // Traverse the top-left diagonal
20             while (topLeftRow >= 0 && topLeftCol >= 0) {
21                 topLeftSet.add(grid[topLeftRow--][topLeftCol--]);
22             }
23
24             // Count distinct values in the top-left diagonal
25             const topLeftDistinctCount = topLeftSet.size;
26
27             // Initialize row and column index for bottom-right diagonal
28             let [bottomRightRow, bottomRightCol] = [i + 1, j + 1];
29             let bottomRightSet = new Set<number>();
30
31             // Create a set to store unique values in the bottom-right diagonal
32             const bottomRightUnique = new Set<number>();
33
34             // Traverse the bottom-right diagonal
35             while (bottomRightRow < rows && bottomRightCol < columns) {
36                 bottomRightSet.add(grid[bottomRightRow++][bottomRightCol++]);
37             }
38
39             // Count distinct values in the bottom-right diagonal
40             const bottomRightDistinctCount = bottomRightSet.size;
41
42             // Calculate the absolute difference of distinct values in both diagonals
43             // and store it in the answer grid for the current cell (i, j)
44             answerGrid[i][j] = Math.abs(topLeftDistinctCount - bottomRightDistinctCount);
45         }
46     }
47
48     // Return the answer grid with the differences
49     return answerGrid;
50 }
51
```

Time and Space Complexity

Time Complexity

The time complexity of the code provided is primarily determined by the two nested loops, each iterating over the dimensions of the grid, and then the while loops inside the nested loops. The outer loops run `n` times where `n` is the number of rows and `c` is the number of columns in the grid (`n` and `c` can be considered separately if not square). The inner while loops iterate at worst from `0` to `min(r, c)` for the top-left diagonal elements, and from `r` to `n` for the bottom-right diagonal elements (and similarly for `c` and `l`). This gives us two arithmetic progressions counting down and up respectively. The sum of operations due to these internal while loops corresponds to the sum of two arithmetic series, which is effectively `O(n * n)` for each outer loop iteration. Therefore, the total time complexity is roughly `O((n * n) + (n * n))`, which can also be represented as `O(n^2 + n * n + n^2)` when `m` and `n` are different.

Space Complexity

The space complexity is `O(n * n)` due to the auxiliary space required for the `ans` list which is the same size as the `grid`. Additionally, the set `s` takes up space, but since it is overwritten in each iteration of the loops rather than accumulated, the maximum space it uses at any given time is the size of the largest number of distinct elements on a diagonal, which is at most `min(n, m)`. This does not change the overall space complexity, which remains `O(n * n)`.