2768. Number of Black Blocks

Medium Array Hash Table Enumeration

Leetcode Link

Problem Description

In the given problem, we are tasked with assessing a grid with dimensions `m x n` that is partially colored with black cells at specific coordinates. Initially, we assume that the entire grid is white. A matrix called `coordinates` is provided to us, where each element `[x, y]` of this matrix represents a cell in our grid that is black.

What we are interested in is identifying the number of 'blocks' within this grid containing a certain number of black cells. A 'block' is defined as a `2 x 2` section of the grid and may therefore contain between zero and four black cells. In essence, our goal is to count the number of these blocks based on the number of black cells they have, ranging from zero to four.

We are expected to return an array of size five, `arr`, where `arr[i]` corresponds to the number of blocks containing exactly `i` black cells. To put this into perspective, `arr[0]` will represent the number of blocks with no black cells, and it progresses all the way to `arr[4]`, which represents the number of blocks completely filled with black cells.

Intuition

The intuition behind the solution involves two primary steps: identifying relevant blocks and counting the number of black cells within each block.

Given that a block is defined by its top-left cell, we can consider each black cell's contribution to possibly four different blocks - the ones that could contain this black cell either in their top-left, top-right, bottom-left, or bottom-right position.

To achieve this, we create a hash table (or a dictionary in Python) where the keys correspond to the top-left coordinates of all potential blocks, and the values represent the number of black cells within each block.

As we iterate through each black cell given in the `coordinates`, we increment the count for the blocks it is part of. We do this carefully, taking into account the edge cases: a black cell that lies on the bottom or right edge of the grid won't be the top-left cell of any block.

The answer to our problem is partially embedded in the hash table - it tells us how many blocks have one or more black cells. However, we still need to account for the blocks that have no black cells at all. To calculate this, we subtract the total number of blocks with at least one black cell (which is just the number of keys in our hash table) from the total number of possible blocks, `(m - 1) * (n - 1)`.

At the end, the `ans` array is assembled by iterating through the hash table's values and incrementing the count in the array based on the number of black cells in each block.

Solution Approach

The solution employs a hash table to track the number of black cells within each potential block. A hash table is a data structure that allows us to store key-value pairs, where the key is unique. In this case, the keys are tuples representing the top-left corner of each block `(x, y)`, and the values are counts of how many black cells are contained within that block.

We iterate over each black cell given in the `coordinates`. For each cell `(x, y)`, we consider its contribution to the surrounding blocks. The surrounding blocks that the cell contributes to are identified by shifting the cell's position one step up/to the left (`(x - 1, y - 1)`, `(x - 1, y)`, `(x, y - 1)`, `(x, y)`). When the cell is on the edge of the grid, some of these shifts would result in invalid coordinates for a block's top-left corner, so we verify that each corner coordinate is valid by checking the grid's bounds.

The Python `Counter` class is utilized to implement the hash table, which already provides a convenient way to increment the count for each key. As we visit each black cell, we update the counts accordingly.

After accounting for all black cells and their contributions, we are left with a hash table where the keys are the possible upper-left corners of `2 x 2` blocks, and the values are the number of black cells in those blocks. We construct the `ans` array from this data, where for each key in the hash table, the value associated with it tells us how many black cells that block contains. We increment the corresponding index in `ans` for the count of such blocks.

Lastly, to calculate the number of blocks that contain zero black cells, we use the formula `(m - 1) * (n - 1) - sum(cnt.values())`. This takes the total possible number of blocks in our grid, subtracts the number of non-zero entries from our hash table, and gives us the number of white blocks.

By iterating through the values of our hash table and assembling the `ans` array, we complete the solution and return it as our answer, adhering to the problem's constraints and expectations.

Example Walkthrough

Let's walk through a small example using the solution approach described above.

Example Grid (m x n): 3 x 3 grid where m and n are the dimensions of the grid

Coordinates of Black Cells: `[[1, 1], [1, 2], [2, 1]]`

Our grid looks like this initially, where `0` represents white cells:

```
1  0 0 0
2  0 0 0
3  0 0 0
```

After coloring the cells from the coordinates, our grid looks like this (`1` represents black cells):

```
1  0 0 0
2  0 1 1
3  0 1 0
```

Let's identify potential blocks. A `2 x 2` block is defined by its top-left corner. In a `3 x 3` grid, the potential top-left corners are (0, 0), (0, 1), (1, 0), and (1, 1).

Now we iterate over each black cell and update the count for each potential block it is part of. We only increment the count if the coordinate is a valid top-left corner of a block.

For black cell (1, 1):

- It will contribute to blocks with top-left corners at (1, 1), (1, 0), (0, 1), and (0, 0). However, only (1, 1) and (0, 1) are valid blocks since (1, 0) and (0, 0) do not form a complete `2 x 2` block within the grid.

For black cell (1, 2):

- It contributes to blocks with top-left corners at (1, 2), (1, 1), (0, 2), and (0, 1). Only (1, 1) and (0, 1) are valid.

For black cell (2, 1):

- It contributes to blocks with top-left corners at (2, 1), (2, 0), (1, 1), and (1, 0). Only (1, 1) is a valid top-left corner for a block.

At this point, the hash table (or Counter) will contain:

- (0, 1): 1 black cell
- (1, 1): 3 black cells

We have only two non-empty blocks; the rest (`(m - 1) * (n - 1) = 4`) are empty blocks.

To calculate the final array, `arr`, we do the following:

- `arr[0]`: Number of white blocks (`(m - 1) * (n - 1)` - number of non-empty blocks) = 4 - 2 = 2.
- `arr[1]`: The number of blocks with exactly 1 black cell = 1 (block with the top-left corner at (0, 1)).
- `arr[2]`: The number of blocks with exactly 2 black cells = 0.
- `arr[3]`: The number of blocks with exactly 3 black cells = 1 (block with the top-left corner at (1, 1)).
- `arr[4]`: The number of blocks with exactly 4 black cells = 0.

Therefore, the final returned array will be: `arr = [2, 1, 0, 1, 0]`. This means there are 2 blocks with 0 black cells, 1 block with 1 black cell, 0 blocks with 2 black cells, 1 block with 3 black cells, and 0 blocks with 4 black cells.

Python Solution

```python
from collections import Counter
from itertools import pairwise

class Solution:
    def countBlackBlocks(self, m: int, n: int, coordinates: List[List[int]]) -> List[int]:
        # Counter to keep track of the number of occurrences for each block
        block_counter = Counter()

        # Iterate through each coordinate
        for x, y in coordinates:
            # Generate 4 adjacent positions using pairwise
            for delta_x, delta_y in pairwise((0, 0, -1, -1, 0)):
                # New coordinates for the adjacent position
                i, j = x + delta_x, y + delta_y

                # Check if the new coordinates are inside the grid boundaries
                if 0 <= i < m - 1 and 0 <= j < n - 1:
                    # Increment the count for this block
                    block_counter[(i, j)] += 1

        # List to store the result; there can be 0 to 4, total 5 different counts
        result_counts = [0] * 5

        # Update the result list with the counts from the counter
        for count in block_counter.values():
            result_counts[count] += 1

        # Calculate the number of blocks with 0 count (not modified by any operation)
        result_counts[0] = (m - 1) * (n - 1) - sum(block_counter.values())

        return result_counts

```

Java Solution

```java
class Solution {
    public long[] countBlackBlocks(int rows, int cols, int[][] coordinates) {
        // A map to keep count of shared corners for each black block
        Map<Long, Integer> sharedCornersCount = new HashMap<>(coordinates.length);
        // Array to navigate through the neighboring blocks
        int[] directions = {0, 0, -1, -1, 0};

        // Iterate through the coordinates of the black blocks
        for (int[] coordinate : coordinates) {
            int x = coordinate[0], y = coordinate[1];
            // Consider the neighboring cells, as they share corners with the current cell
            for (int k = 0; k < 4; k++) {
                int i = x + directions[k], j = y + directions[k + 1];
                // Check if the neighboring cell is within the boundaries of the grid
                if (i >= 0 && i < rows - 1 && j >= 0 && j < cols - 1) {
                    // Using long to avoid integer overflow for indexing
                    long index = 1L * i * cols + j;
                    // Count the shared corners for each cell
                    sharedCornersCount.merge(index, 1, Integer::sum);
                }
            }
        }

        // Array to hold the count of cells with a specific number of shared corners
        long[] result = new long[5];
        // By default, all the cells are initialized with no shared corners
        result[0] = (long)(rows - 1) * (cols - 1);
        // Iterate through the counts and update the result array accordingly
        for (int count : sharedCornersCount.values()) {
            result[count]++;      // Increment the count for the specific shared corners
            result[0]--;          // Decrement the count of cells with no shared corners
        }

        return result;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>

class Solution {
public:
    std::vector<long long> countBlackBlocks(int m, int n, std::vector<std::vector<int>>& coordinates) {
        // A map to keep count of the black blocks shaded by each coordinate
        std::unordered_map<long long, int> block_counts;

        // Directions representing the relative positions of the 4 neighboring blocks
        int directions[5] = {0, 0, -1, -1, 0};

        // Loop through each coordinate in the coordinates list
        for (const auto& coordinate : coordinates) {
            int x = coordinate[0], y = coordinate[1];
            // Check all four neighboring positions
            for (int k = 0; k < 4; ++k) {
                int neighbor_x = x + directions[k], neighbor_y = y + directions[k + 1];
                // Check if the neighbor block is within bounds
                if (neighbor_x >= 0 && neighbor_x < m - 1 && neighbor_y >= 0 && neighbor_y < n - 1) {
                    // Increment the count for the neighbor block key in the map
                    ++block_counts[static_cast<long long>(neighbor_x) * n + neighbor_y];
                }
            }
        }

        // Answer vector containing counts of black blocks shaded 0 to 4 times
        std::vector<long long> answer(5);
        // Initially set the count of blocks not shaded to the total number of possible blocks
        answer[0] = static_cast<long long>(m - 1) * (n - 1);
        // Update the answer vector based on the counts in the map
        for (const auto& count : block_counts) {
            // Increment the count for the number of times a block is shaded
            ++answer[count.second];
            // Decrement the count of blocks not shaded
            --answer[0];
        }

        // Return the final answer vector
        return answer;
    }
};

```

Typescript Solution

```typescript
function countBlackBlocks(m: number, n: number, coordinates: number[][]): number[] {
    // Map to keep track of the count of blocks at each grid position
    const blockCount: Map<number, number> = new Map();
    // Directions to move to adjacent cells (left, right, up, down)
    const directions: number[] = [0, 0, -1, -1, 0];

    // Iterate over each coordinate
    for (const [row, col] of coordinates) {
        // Check each direction from the coordinate
        for (let k = 0; k < 4; ++k) {
            // Calculate the adjacent cell's position
            const [adjRow, adjCol] = [row + directions[k], col + directions[k + 1]];
            // Ensure that the adjacent cell is within the grid bounds
            if (adjRow >= 0 && adjRow < m - 1 && adjCol >= 0 && adjCol < n - 1) {
                // Calculate a unique key for the position in the grid
                const key = adjRow * n + adjCol;
                // Increment the block count for this key, or set to 1 if it doesn't exist
                blockCount.set(key, (blockCount.get(key) || 0) + 1);
            }
        }
    }

    // Array to store the final count of black blocks for each count type (0 to 4)
    const answer: number[] = Array(5).fill(0);
    // Initialize the answer for count type 0 to the total possible number of grid cells
    answer[0] = (m - 1) * (n - 1);

    // Iterate over the block count map to populate the answer array
    for (const count of blockCount.values()) {
        // Increment the answer for the corresponding count type
        ++answer[count];
        // Decrement the answer for count type 0 since we've found a cell with >0 blocks
        --answer[0];
    }
    return answer;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is primarily dominated by the for loop that iterates through each of the `coordinates`. Within this loop, there's a call to `pairwise`, which will have a constant number of iterations (specifically, 2 in this case) since it is paired with a fixed size tuple. Therefore, the complexity due to `pairwise` does not depend on the size of the input.

Next, we have a nested for loop, but since it also iterates over a fixed-size tuple (four elements), it runs in constant time per coordinate.

Given that each iteration of the main for loop is constant time and we have `l` coordinates, the overall time complexity is `O(l)`.

Space Complexity

The space complexity is influenced by the `Counter` object `cnt` that collects the frequency of each black block. In the worst-case scenario, each coordinate touches a unique block, thus the space required to store counts can grow linearly with the number of coordinates. Therefore, the space complexity is also `O(l)`, where `l` is the length of `coordinates`.

Note that the output array `ans` has a fixed size of 5, and the fixed tuples used for iteration do not scale with input size, so they do not affect the space complexity.