542. 01 Matrix

Medium | Breadth-First Search | Array | Dynamic Programming | Matrix | Leetcode Link

Problem Description

The task is to find the distance to the nearest 0 for each cell in a given `m x n` binary matrix `mat`. A binary matrix is a matrix where each cell can either be a 0 or a 1. The distance is defined as the minimum number of adjacent cell steps (up, down, left, or right) needed to reach a cell with the value 0 from the current cell. For example, if you're moving from `mat[i][j]` to `mat[i][j+1]` or `mat[i+1][j]`, that counts as a distance of 1. The problem requires us to populate and return a new matrix with the same dimensions as the input matrix, where each cell contains the distance to the nearest 0 in the original matrix.

Intuition

To solve this problem, we can use the Breadth-First Search (BFS) algorithm. The key intuition is that the BFS allows us to find the shortest path in unweighted graphs (or matrices, in this case), starting from a source node (or cell with a value of 0). The steps are as follows:

1. We first create an auxiliary matrix (`ans`) of the same dimensions as the input matrix `mat`, where each cell is initialized to -1 except the ones corresponding to 0s in the original matrix, which are initialized to 0. This new matrix will eventually hold the distance of each cell to the nearest 0.

2. We then use a queue to keep track of cells that we need to explore. Initially, we enqueue the coordinates of all cells with 0s, as their distance to the nearest 0 is already known (which is 0).

3. Now we perform the BFS. We dequeue a cell `(i, j)` from the queue and look at its immediate neighbors (up, down, left, right). If a neighbor `(x, y)` has not been visited before (which we check by seeing if `ans[x][y]` is -1), we set its value to be one more than the dequeued cell (`ans[i][j] + 1`) since it's an adjacent cell.

4. After updating the distance of a neighboring cell, we enqueue it to the queue to later inspect its neighbors too.

5. This process continues until the queue is empty, meaning all cells have been visited and assigned the distance to the nearest 0.

By the end of this BFS, the `ans` matrix will contain the distances of all cells to their nearest 0 cell, as desired.

Solution Approach

The solution provided follows a multi-source Breadth-First Search (BFS) approach. Here's a step-by-step walkthrough:

1. **Matrix Initialization**: An auxiliary matrix `ans` with the same dimensions as the input matrix `mat` is created. Each cell in `ans` is initialized to `-1`, except for the cells that correspond to `0` in `mat`; these are set to `0`. The reason for using `-1` is to indicate that a cell's distance to the nearest `0` hasn't been calculated yet.

2. **Queue Initialization**: A queue named `q` is used to store the cells which distances we want to calculate. All the positions of cells that contain `0` in the original matrix `mat` are put into the queue first. These serve as the starting points for the BFS since they are already `0` and don't need their distances calculated.

3. **BFS Implementation**: The algorithm dequeues an element `(i, j)` from `q` and looks at each of its four neighboring cells. By using the `dirs` tuple, which is `(-1, 0, 1, 0, -1)`, we can consider all four directions - up, down, left, and right by pairing `dirs` elements using the `pairwise` utility (this utility pairs elements such that we get pairs for all adjacent directions).

4. **Neighbor Inspection and Update**:

   - It then iterates through these adjacent cells `(x, y)` and checks if each is within the bounds of the matrix and if `ans[x][y]` is still set to `-1`.
   - If these conditions are met, we know we're looking at a cell whose shortest distance to a `0` hasn't been calculated yet.
   - The distance is then set to be `1` more than its neighbor `(i, j)` from which it was reached (`ans[i][j] + 1`).
   - The position `(x, y)` is then enqueued, which means its neighbors will be examined in the next iterations of BFS.

5. **Termination**: The BFS process continues by repeatedly dequeuing, inspecting, updating, and enqueuing until the queue `q` is empty. At that point, the matrix `ans` contains the minimum distances of all cells to the nearest `0`.

The use of BFS is key here because it guarantees that we update each cell with the minimum distance to `0`. This happens because in BFS, we start updating distances from `0`s themselves and move outward, ensuring that the first time a cell's distance is calculated, it is the shortest path. If we used Depth-First Search (DFS), we could end up visiting cells multiple times to update their distances, which would be inefficient.

Lastly, this implementation works efficiently since each cell is enqueued at most once, leading to a time complexity that is linear in the number of cells in the matrix, which is `O(m*n)` where `m` and `n` are the dimensions of the input matrix.

Example Walkthrough

Given a binary matrix:

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

Let's walk through the solution approach using this matrix.

1. **Matrix Initialization**: Create `ans` matrix of the same dimensions with `-1` for all cells except cells corresponding to `0` in `mat`:

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

2. **Queue Initialization**: Initialize the queue `q` with the coordinates of all `0`s from `mat`:

```
q = [(0, 0), (0, 1), (2, 2)]
```

3. **BFS Implementation**: Since BFS uses a queue, we start by taking the first element in `q`, which is `(0, 0)` and explore its neighbors to see if their `ans` values need to be updated.

4. **Neighbor Inspection and Update**:

   - For the element `(0, 0)` from the queue, it has two neighbors `(0, 1)` and `(1, 0)`. Since `(0, 1)` is already `0`, we skip it. No updates are needed as it's already processed.
   - Look at neighbor `(1, 0)`. It's within the bounds and `ans[1][0]` is `-1`, thus it hasn't been visited. We update `ans[1][0]` to `1` (as `ans[0][0] + 1`) and enqueue `(1, 0)`:

```
ans = [
  [0, 0, -1],
  [1, -1, -1],
  [-1, -1, 0]
]
q = [(0, 1), (2, 2), (1, 0)]
```

5. **Continue BFS**: Follow the BFS algorithm until `q` is empty, updating `ans` as you go. The final iterations will proceed as follows:

   - `(0, 1)` is skipped since it's a `0`.
   - `(2, 2)` is a `0`, its neighbors are inspected and updated accordingly.
   - `(1, 0)` is dequeued next, updating its neighbors.
   - Repeat this process until `q` is empty.

By the end of BFS, `q` is emptied and the `ans` matrix is fully updated:

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

6. **Termination**: The final `ans` matrix now contains the distance to the nearest `0` for each cell, and we return this as the solution.

Python Solution

```python
from collections import deque

class Solution:
    def updateMatrix(self, matrix: List[List[int]]) -> List[List[int]]:
        # Get the matrix dimensions
        rows, cols = len(matrix), len(matrix[0])
        # Prepare an answer matrix with the same dimensions
        distance_matrix = [[-1] * cols for _ in range(rows)]
        # Initialize a queue to store the indices of 0s
        queue = deque()

        # Go through the matrix to find all the 0s
        for i in range(rows):
            for j in range(cols):
                if matrix[i][j] == 0:
                    # Set the distance for 0s as 0
                    distance_matrix[i][j] = 0
                    # Add the coordinates of the 0 to the queue
                    queue.append((i, j))

        # Directions for moving up, left, down, right
        directions = ((-1, 0), (0, -1), (1, 0), (0, 1))

        # Process the queue
        while queue:
            i, j = queue.popleft()
            # Explore all neighboring cells in the four directions
            for delta_i, delta_j in directions:
                neighbor_i, neighbor_j = i + delta_i, j + delta_j
                # If the neighbor is within bounds and hasn't been visited
                if 0 <= neighbor_i < rows and 0 <= neighbor_j < cols and distance_matrix[neighbor_i][neighbor_j] == -1:
                    # Update the distance for the neighbor
                    distance_matrix[neighbor_i][neighbor_j] = distance_matrix[i][j] + 1
                    # Add the neighbor's coordinates to the queue for further exploration
                    queue.append((neighbor_i, neighbor_j))

        # Return the updated distance matrix
        return distance_matrix

```

Java Solution

```java
class Solution {
    public int[][] updateMatrix(int[][] matrix) {
        // Get the dimensions of the matrix.
        int rows = matrix.length, cols = matrix[0].length;

        // Create a matrix to store the answer with the same dimensions.
        int[][] distanceMatrix = new int[rows][cols];

        // Initialize the distance matrix with -1 to mark as not processed.
        for (int[] row : distanceMatrix) {
            Arrays.fill(row, -1);
        }

        // Create a queue to perform the BFS.
        Deque<int[]> queue = new ArrayDeque<>();

        // Loop through every cell in the matrix.
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {

                // If the current cell has a 0, mark same in the distance matrix, and add it to the queue.
                if (matrix[i][j] == 0) {
                    queue.offer(new int[] {i, j});
                    distanceMatrix[i][j] = 0;
                }
            }
        }

        // Array to help iterate over the 4 neighboring cells (up, right, down, left).
        int[] directions = {-1, 0, 1, 0, -1};

        // Perform BFS on the queue until it's empty.
        while (!queue.isEmpty()) {
            // Poll an element from the queue
            int[] position = queue.poll();
            int currentRow = position[0], currentCol = position[1];

            // Iterate over the four possible neighbors of the current cell.
            for (int k = 0; k < 4; ++k) {
                int newRow = currentRow + directions[k], newCol = currentCol + directions[k + 1];

                // Check if the new cell is within bounds and hasn't been visited yet.
                if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && distanceMatrix[newRow][newCol] == -1) {
                    // Update the distance matrix with the distance from the nearest 0.
                    distanceMatrix[newRow][newCol] = distanceMatrix[currentRow][currentCol] + 1;
                    // Add the new cell to the queue to continue the BFS.
                    queue.offer(new int[] {newRow, newCol});
                }
            }
        }

        // Return the updated distance matrix.
        return distanceMatrix;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {
        int rows = matrix.size(); // Number of rows in the matrix
        int cols = matrix[0].size(); // Number of columns in the matrix
        vector<vector<int>> dist(rows, vector<int>(cols, -1)); // Initialize the distance matrix with -1
        queue<pair<int, int>> queue; // Queue to store the matrix cells

        // Initialize the queue with all the cells that have 0s and set their distance to 0
        for (int row = 0; row < rows; ++row) {
            for (int col = 0; col < cols; ++col) {
                if (matrix[row][col] == 0) {
                    dist[row][col] = 0;
                    queue.emplace(row, col);
                }
            }
        }

        // Defining directions for easy access to adjacent cells in matrix (up, right, down, left)
        vector<int> directions = {-1, 0, 1, 0, -1};

        // Perform Breadth-First Search
        while (!queue.empty()) {
            pair<int, int> cur = queue.front(); // Get current cell from the queue
            queue.pop();

            // Check all adjacent cells
            for (int i = 0; i < 4; ++i) { // Loop through all four possible directions
                int nextRow = cur.first + directions[i];   // Compute row index for adjacent cell
                int nextCol = cur.second + directions[i+1]; // Compute column index for adjacent cell

                // Check if the adjacent cell is within bounds and hasn't been visited
                if (nextRow >= 0 && nextRow < rows && nextCol >= 0 && nextCol < cols && dist[nextRow][nextCol] == -1) {
                    // If the cell is valid and unvisited, calculate its distance and add to queue
                    dist[nextRow][nextCol] = dist[cur.first][cur.second] + 1;
                    queue.emplace(nextRow, nextCol);
                }
            }
        }

        return dist; // Return the distance matrix after processing.
    }
};

```

Typescript Solution

```typescript
function updateMatrix(matrix: number[][]): number[][] {
    // Dimension of the given matrix
    const numRows: number = matrix.length;
    const numCols: number = matrix[0].length;

    // Initialize the answer matrix with -1,
    // indicating that distances have not been calculated yet
    const answerMatrix: number[][] = Array.from({ length: numRows }, () =>
        Array(numCols).fill(-1));

    // Queue to maintain the cells with value 0
    const queue: [number, number][] = [];

    // Populate the queue with all the cells that contain 0
    // and update their distance in the answer matrix
    for (let row = 0; row < numRows; ++row) {
        for (let col = 0; col < numCols; ++col) {
            if (matrix[row][col] === 0) {
                queue.push([row, col]);
                answerMatrix[row][col] = 0;
            }
        }
    }

    // Directions array to explore adjacent cells (up, right, down, left)
    const directions: number[] = [-1, 0, 1, 0, -1];

    // Process the queue and update each cell's distance from the nearest 0
    while (queue.length > 0) {
        const [currentRow, currentCol] = queue.shift()!;

        // Explore the adjacent cells in 4 directions
        for (let k = 0; k < 4; ++k) {
            const nextRow = currentRow + directions[k];
            const nextCol = currentCol + directions[k + 1];

            // If the next cell is within bounds and its distance is not calculated
            if (nextRow >= 0 && nextRow < numRows &&
                nextCol >= 0 && nextCol < numCols &&
                answerMatrix[nextRow][nextCol] === -1) {
                // Update the distance and add the cell to the queue
                answerMatrix[nextRow][nextCol] = answerMatrix[currentRow][currentCol] + 1;
                queue.push([nextRow, nextCol]);
            }
        }
    }

    // Return the updated answer matrix
    return answerMatrix;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by the number of elements in the matrix and the operations performed on each of those elements. Here's a breakdown:

- The nested for loops go through each element of the matrix to initialize the `ans` array and populate the queue with the positions of the zeroes, which takes `O(m * n)` time where `m` is the number of rows, and `n` is the number of columns in the matrix.

- The while loop dequeues each position from the queue at most once. Since each element can be added to the queue only once, the while loop will have at most `m * n` iterations.

- Inside the while loop, we iterate over the four possible directions from each position, which is a constant time operation.

Since the number of operations for each element is constant, the overall time complexity of this code is `O(m * n)`.

Space Complexity

The space complexity is determined by the additional space used by the algorithm, not including the space for the input and output:

- The `ans` array, which has the same dimensions as the input matrix, uses `O(m * n)` space.

- The queue `q` can store a maximum of `m * n` positions (in the worst case when all elements are zero), which also takes `O(m * n)` space.

Therefore, the overall space complexity of the algorithm is `O(m * n)`.