1030. Matrix Cells in Distance Order

Easy Geometry Array Math Matrix Sorting   Leetcode Link

Problem Description

You are tasked with finding the coordinates of all cells in a given matrix, ordered by their distance from a specified center cell. Given four integers (`row`, `cols`, `rCenter`, `cCenter`), the problem defines a matrix of size `rows x cols`. Your position in the matrix is initially at the cell with coordinates `(rCenter, cCenter)`.

The goal is to return a list of coordinates of all cells in the matrix, sorted based on their Manhattan distance from the center cell `(rCenter, cCenter)`. The Manhattan distance between two points `(r1, c1)` and `(r2, c2)` is calculated as `|r1 - r2| + |c1 - c2|`. In other words, it's the sum of the absolute differences of their corresponding coordinates.

You have the flexibility to provide the sorted coordinates in any order as long as they are arranged from the nearest to the farthest distance from the center cell.

Intuition

The intuition behind the solution to this problem lies in the properties of the Manhattan distance and an algorithmic technique known as Breadth-First Search (BFS).

Manhattan distance has an interesting property where cells that are an equal distance from a given point form a diamond shape when graphed on the matrix. Since the problem asks us to order the cells by their distance from a center cell, using the BFS algorithm, we can ensure that we process cells in layers - starting with the center cell and moving outwards one distance unit at a time.

Therefore, the approach is to create a queue and start from the center cell, adding it to the queue. We repeatedly take a cell from the front of the queue, add it to our answer list, and then add all its unvisited neighboring cells that are one unit away. To avoid adding the same cell multiple times, we also keep track of all visited cells using a boolean matrix called `vis`.

This method ensures that we add cells to our answer list in increasing distance order without explicitly calculating the distance for every cell from the center. Once there are no more cells to add, we have processed the whole matrix, and our answer list contains all cells in the required sorted order.

Solution Approach

The implementation of the solution uses Breadth-First Search (BFS) to explore the matrix starting from the given center. Here's a step-by-step walkthrough explaining how the given Python code achieves this:

1. **Initialize Data Structures**: A queue `q` is used to store cells for exploration, and it's implemented using a `deque` to allow efficient popping from the front. A 2-dimensional list `vis` keeps track of visited cells; initially, all cells are marked as unvisited (`False`).
2. **Queue Starting Cell**: The starting cell, given by the coordinates `(rCenter, cCenter)`, is added to the queue and marked as visited in the `vis` matrix.
3. **Process Cells in Queue**: The algorithm enters a while-loop that continues until the queue is empty. At each step, we process all cells currently in the queue, ensuring that all neighbors one unit away are considered before moving to a larger distance.
4. **Explore Neighbors**: Within the loop, for each cell `(p)` taken from the queue, its neighbors are determined by iterating over the relative positions represented by `(-1, 0, 1, 0, -1)` using the `pairwise` function which gives us pairs of relative coordinates for the four directions (up, right, down, left). These are used to calculate the neighboring cells' coordinates `(x, y)`.
5. **Boundary and Visit Checks**: Before a neighboring cell is added to the queue, there are two checks:
   - **Boundary Check**: The coordinates `(x, y)` must be inside the matrix, which is checked with `0 <= x < rows` and `0 <= y < cols`.
   - **Visit Check**: The `vis` matrix is checked to ensure the cell has not been visited before. If it hasn't, it is marked as visited.
6. **Enqueue and Store Results**: If a neighbor passes these checks, it is appended to the queue for subsequent exploration and also added to the growing result set `ans`.
7. **Return Results**: After the `while` loop exits (when the queue is empty), it means that all cells have been explored according to their distance from the center, and the `ans` list contains them in the order of increasing Manhattan distance. The list `ans` is returned as the final output.

The algorithm efficiently traverses the matrix in a manner that naturally sorts the cells by their Manhattan distance, eliminating the need for an explicit sort operation, and is a classic example of BFS applied to grid traversal problems.

Example Walkthrough

Let's illustrate the solution approach using a small example. Suppose our matrix size is defined by `rows = 3` and `cols = 3`, creating a 3×3 grid. The center cell is given by `rCenter = 1` and `cCenter = 1`. The matrix with Manhattan distance from the center cell looks like this:

```
1 1 1
1 0 1
1 1 1
```

Initialization:

- Queue `q` is initialized and contains the starting cell coordinates `(1, 1)`.
- Visited matrix `vis` is initialized with all values set to `False`.

Queue Starting Cell:

- Starting cell `(1, 1)` is marked as visited in `vis` and added to `q`.

Process Cells in Queue:

- We begin processing by popping `(1, 1)` from the queue.

Explore Neighbors:

- The neighbors of `(1, 1)` are `(1, 0)`, `(2, 1)`, `(1, 2)`, `(0, 1)`.

Boundary and Visit Checks:

- All neighbors are within the boundaries of the matrix and none of them are visited. They are marked as visited and added to the queue `q`.

Enqueue and Store Results:

- The order of neighbors being added to `q` and result set `ans` may vary, but let's assume they're added in the sequence `(1, 0)`, `(2, 1)`, `(1, 2)`, `(0, 1)`.

Return Results:

- The next level of neighbors would be the corners, `(0, 0)`, `(0, 2)`, `(2, 0)`, `(2, 2)`, each with Manhattan distance of 2.
- The process continues until all cells have been added to `ans`.

In the end, `ans` would look like `[(1, 1), (1, 0), (2, 1), (1, 2), (0, 1), (0, 0), (0, 2), (2, 0), (2, 2)]` or any other such sequence that preserves the non-decreasing Manhattan distance order.

This demonstrates how the BFS approach efficiently processes each layer of cells, grouped by their Manhattan distance to the center cell, until all cells have been visited and added to the result in the required sorted order.

Python Solution

```python
from collections import deque
from typing import List

class Solution:
    def allCellsDistOrder(self, rows: int, cols: int, r_center: int, c_center: int) -> List[List[int]]:
        # Initialize a queue with the starting cell, which is the center cell
        queue = deque([[r_center, c_center]])
        # Create a 2D list to keep track of visited cells
        visited = [[False] * cols for _ in range(rows)]
        visited[r_center][c_center] = True

        # List to store the cells in the order of increasing distance from the center
        result = []

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

        # Perform a Breadth-First Search (BFS) starting from the center cell
        while queue:
            # Dequeue the front cell of the queue
            current_cell = queue.popleft()
            result.append(current_cell)

            # Try moving in all four directions from the current cell
            for delta_row, delta_col in directions:
                new_row = current_cell[0] + delta_row
                new_col = current_cell[1] + delta_col

                # Check if the new cell is within grid bounds and hasn't been visited
                if 0 <= new_row < rows and 0 <= new_col < cols and not visited[new_row][new_col]:
                    # Mark the new cell as visited
                    visited[new_row][new_col] = True
                    # Add the new cell to the queue to explore its neighbors later
                    queue.append([new_row, new_col])

        # Return the cells in the order they were visited
        return result

```

Java Solution

```java
import java.util.ArrayDeque;
import java.util.Deque;

class Solution {
    // Method to return the coordinates of all cells in the matrix, sorted by their distance from (rCenter, cCenter)
    public int[][] allCellsDistOrder(int rows, int cols, int rCenter, int cCenter) {
        // Initialize a queue to perform the breadth-first search
        Deque<int[]> queue = new ArrayDeque<>();
        // Start from the center cell
        queue.offer(new int[] {rCenter, cCenter});

        // Create a visited matrix to keep track of visited cells
        boolean[][] visited = new boolean[rows][cols];
        // Mark the center cell as visited
        visited[rCenter][cCenter] = true;

        // Create a result array to hold all cells in the required order
        int[][] result = new int[rows * cols][2];
        // Use the `dirs` array to explore in all four directions
        int[] dirs = {-1, 0, 1, 0, -1};
        // Index to insert the next point in `result`
        int index = 0;

        // Perform breadth-first search
        while (!queue.isEmpty()) {
            for (int size = queue.size(); size > 0; size--) {
                // Get the current cell from the queue
                int[] point = queue.poll();
                // Assign the current cell's coordinates to the result array
                result[index++] = point;

                // Explore the neighbors of the current cell
                for (int k = 0; k < 4; ++k) {
                    int x = point[0] + dirs[k], y = point[1] + dirs[k + 1];
                    // Check for valid boundary conditions and unvisited state
                    if (x >= 0 && x < rows && y >= 0 && y < cols && !visited[x][y]) {
                        // Mark the new cell as visited
                        visited[x][y] = true;
                        // Add new cell's coordinates to the queue
                        queue.offer(new int[] {x, y});
                    }
                }
            }
        }
        // Return the result array containing all cell coordinates
        return result;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <queue>
#include <cstring>

class Solution {
public:
    vector<vector<int>> allCellsDistOrder(int rows, int cols, int rCenter, int cCenter) {
        // Queue to perform BFS
        queue<pair<int, int>> queue;
        queue.emplace(rCenter, cCenter); // Start from the center cell

        // Initialize answer vector
        vector<vector<int>> answer;

        // Visited matrix to keep track of visited cells
        bool visited[rows][cols];
        memset(visited, false, sizeof(visited)); // Set all cells to unvisited
        visited[rCenter][cCenter] = true; // Mark the center cell as visited

        // Array to easily access all 4 surrounding cells (up, right, down, left)
        int directions[5] = {-1, 0, 1, 0, -1};

        // Perform BFS
        while (!queue.empty()) {
            int queueSize = queue.size(); // Number of elements at current level
            while (queueSize--) {
                auto [currentRow, currentCol] = queue.front();
                queue.pop();

                // Add the current cell to the answer
                answer.push_back({currentRow, currentCol});

                // Explore the neighboring cells
                for (int k = 0; k < 4; ++k) {
                    int newRow = currentRow + directions[k];
                    int newCol = currentCol + directions[k + 1];

                    // Check if the new cell is within bounds and not visited
                    if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && !visited[newRow][newCol]) {
                        visited[newRow][newCol] = true; // Mark cell as visited
                        queue.emplace(newRow, newCol); // Add the cell to the queue for further BFS
                    }
                }
            }
        }

        return answer; // Return the cells sorted by their distance from the center
    }
};

```

Typescript Solution

```typescript
type Cell = [number, number];  // Defines a type for cells

// Define a function to calculate all cells in distance order
function allCellsDistOrder(rows: number, cols: number, rCenter: number, cCenter: number): Cell[] {
    // Queue to perform BFS
    const queue: Cell[] = [[rCenter, cCenter]];

    // Initialize answer array
    const answer: Cell[] = [];

    // Visited matrix to keep track of visited cells. Initialize with false values.
    const visited: boolean[][] = Array.from({ length: rows }, () => Array(cols).fill(false));

    // Mark the center cell as visited
    visited[rCenter][cCenter] = true;

    // Array to easily access all 4 adjacent cells (up, right, down, left)
    const directions: number[] = [-1, 0, 1, 0, -1];

    // Perform BFS
    while (queue.length > 0) {
        const [currentRow, currentCol] = queue.shift()!; // Get the first cell in the queue ! non-null assertion operator since Arr

        // Add the current cell to the answer
        answer.push([currentRow, currentCol]);

        // Explore the neighboring cells
        for (let k = 0; k < 4; k++) {
            const newRow: number = currentRow + directions[k];
            const newCol: number = currentCol + directions[k + 1];

            // Check if the new cell is within bounds and not visited
            if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && !visited[newRow][newCol]) {
                visited[newRow][newCol] = true; // Mark cell as visited
                queue.push([newRow, newCol]); // Add the cell to the queue for further BFS
            }
        }
    }

    // Return the cells sorted by their distance from the center
    return answer;
}

```

Time and Space Complexity

Time Complexity

The given code uses a BFS (Breadth-First Search) approach to traverse the matrix starting from the center cell `(rCenter, cCenter)`. For each cell, it checks all four adjacent cells (up, down, left, right), which are represented by the pairwise combinations `(-1, 0, 1, 0, -1)`.

The time complexity of this approach is `O(R * C)`, where `R` is the number of rows and `C` is the number of columns in the matrix. This is because each cell is visited exactly once. The check `0 <= x < rows and 0 <= y < cols` happens for 4 neighbors for each of the `R * C` cells, but since each neighbor is only enqueued once (guarded by `vis[x][y]`), the total number of operations is still proportional to the number of cells.

Space Complexity

The space complexity of the code is also `O(R * C)`. The main factors contributing to the space complexity are:

1. The `vis` array, which is a 2D array used to keep track of visited cells, consuming `R * C` space.
2. The queue `q`, which in the worst case may contain all cells before being dequeued, thus also requiring up to `R * C` space.
3. The `ans` array, which will eventually hold all `R * C` cells in the order they were visited.

Therefore, the space required is proportional to the number of cells in the matrix.