1926. Nearest Exit from Entrance in Maze

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

Problem Description

In this problem, we are given a maze represented by a 2D matrix. Each cell of the matrix can either be an empty cell (`'.'`) or a wall (`'+'`). An entrance to the maze is also given to us, specified by its row and column index. We are asked to find the shortest path from the entrance to the nearest exit of the maze. Here, an exit is defined as an empty cell located on the border of the maze, excluding the entrance itself. In each step, we can move to a cell that is adjacent (left, right, up, or down) to our current position, but we cannot move into walls or outside the maze boundaries.

Our goal is to navigate from the entrance to the nearest border cell (exit) by taking the fewest steps possible. If we can't find any path to an exit, we must return `-1`. The number of steps is counted as the minimum number of moves required to reach any exit from the entrance.

Intuition

To solve this problem, we can use the Breadth-First Search (BFS) algorithm. BFS is an ideal choice for this kind of problem because it explores all possible paths level by level or, in this case, step by step from the entrance. As a result, the first time it reaches an exit, it is guaranteed to be the nearest one since BFS doesn't explore deeper paths until all paths of the current depth are explored.

We initialize BFS from the entrance by marking it as visited (to avoid revisiting) and then iteratively exploring all four adjacent cells. If an adjacent cell is empty and within the maze bounds, we check if it's an exit. If it's an exit, we immediately return the current step count since it's the minimum. If it's not, we continue the BFS by adding the cell to the queue. Importantly, as we enqueue a cell, we mark it with a wall to avoid revisiting cells that are already considered, effectively reducing unnecessary calculations.

If the BFS completes without finding an exit, we conclude that no path exists, and we return `-1`, indicating failure to reach an exit.

Solution Approach

The solution provided uses the Breadth-First Search (BFS) algorithm to traverse through the maze. Let's dissect the given Python code to better understand how it translates the BFS strategy into a working solution.

```python
class Solution:
    def nearestExit(self, maze: List[List[str]], entrance: List[int]) -> int:
        m, n = len(maze), len(maze[0])  # Dimensions of the maze
        i, j = entrance  # Starting position (entrance of the maze)
        q = deque([(i, j)])  # Initialize the queue with the entrance
        maze[i][j] = '+'  # Mark the entrance as visited by replacing it with '+'
        ans = 0  # Steps counter
        while q:
            ans += 1  # Increment the step counter at the beginning of each level
            for _ in range(len(q)):  # Loop over every node in the current level
                i, j = q.popleft()  # Dequeue the front element from the queue
                for a, b in [[0, -1], [0, 1], [-1, 0], [1, 0]]:  # The 4 possible directions
                    x, y = i + a, j + b  # Calculate the next cell's coordinates
                    if 0 <= x < m and 0 <= y < n and maze[x][y] == '.':  # Check if it's within bounds and not visited
                        if x == 0 or x == m - 1 or y == 0 or y == n - 1:  # Check if it's an exit
                            return ans  # Return the current step count as the answer
                        q.append((x, y))  # Enqueue the cell for future exploration
                        maze[x][y] = '+'  # Mark the cell as visited
        return -1  # If the loop ends without finding an exit, return -1
```

Key Elements of the Solution:

- **Queue (`deque`)**: A queue is used for the BFS traversal, which follows the First-In-First-Out (FIFO) principle. This ensures that cells are explored in the order they are reached.
- **Visited Marking**: When a cell is visited, it is marked by changing its value to `'+'`. This prevents the algorithm from re-visiting the same cell, which would otherwise lead to infinite loops and incorrect step count.
- **Direction Array**: A simple 2D array `[[0, -1], [0, 1], [-1, 0], [1, 0]]` is used to represent the four possible moves from any cell (left, right, up, down).
- **Checking Exit Condition**: An exit is an empty cell (`'.'`) at the border of the maze. Whenever moving to a new cell, the algorithm checks if it is an exit by comparing its coordinates with the boundary of the maze.
- **Steps Counter**: The variable `ans` is used to count the number of steps taken to reach an exit. It gets incremented at the start of processing each level, ensuring the correct step count when an exit is found.

Efficiency of the Solution:

The solution is efficient for finding the shortest path in a maze scenario. BFS ensures that the first time we find an exit, it must be the closest one because we explore all possible paths of equal length before moving on to longer paths. The time and space complexity of this solution are both O(m*n), where `m` is the number of rows and `n` is the number of columns in the maze.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider the following 5×5 maze as an example, where `E` is the entrance, `.` represents an empty cell, and `+` represents a wall:

```
+ + + + +
+ . . . +
+ . E . +
+ . . . +
+ + + . +
```

Following the BFS strategy to navigate the maze from the entrance to the closest exit:

1. Initialize the queue with the entrance coordinates, which is `(2, 2)` here, and set the starting cell as visited by marking it with '+':

```
+ + + + +
+ . . . +
+ . + . +
+ . . . +
+ + + . +
```

2. Process the first node in the queue, we explore its adjacent cells: `(2, 1)`, `(2, 3)`, `(1, 2)`, and `(3, 2)`. These cells are not yet visited; we add them to the queue and mark them as visited:

```
+ + + + +
+ . + . +
+ + + . +
+ . + . +
+ + + . +
```

3. We increment our steps counter `ans = 1` as we have started moving from the entrance. Now we process the cells in the queue. For each cell, we check its adjacent cells. For example, start with cell `(2, 1)` and check `(1, 1)`, `(3, 1)`, `(2, 0)`, and `(2, 2)`. Here, `(2, 0)` is not within the bounds, `(2, 2)` is already visited, and `(1, 1)` and `(3, 1)` are available for further exploration.
4. As we continue to explore the current level, we find that the cell `(1, 2)` is just one step away from an exit as it is located on the border of the maze. We've reached an exit:

```
+ + + + +
+ . + . +
+ + + . +
+ . + . +
+ + + . +
```

5. Since we've encountered an exit, we immediately return the current step count, which is `ans = 1`.

By the BFS approach, we've managed to find the nearest exit to the entrance in the fewest steps as possible without unnecessary calculations. If no exit had been reached, we would have continued to process the BFS queue until it was empty and then returned `-1` to indicate no available exit.

Python Solution

```python
from collections import deque

class Solution:
    def nearest_exit(self, maze, entrance):
        # Get the dimensions of the maze
        rows, cols = len(maze), len(maze[0])

        # Entrance coordinates
        row_entrance, col_entrance = entrance

        # Initialize a queue with the starting position (entrance)
        queue = deque([(row_entrance, col_entrance)])

        # Mark the starting position as visited by changing it to '+'
        maze[row_entrance][col_entrance] = '+'

        # Initialize the number of steps taken to exit
        steps = 0

        # Breadth-first search (BFS) loop
        while queue:
            # Increment the steps at the start of each level of BFS
            steps += 1

            # Go through each position at the current level
            for _ in range(len(queue)):
                # Get position from queue
                row, col = queue.popleft()

                # Directions in which we can move: left, right, up, down
                directions = [[0, -1], [0, 1], [-1, 0], [1, 0]]

                # Check all four directions
                for direction_row, direction_col in directions:
                    # Calculate new position
                    new_row, new_col = row + direction_row, col + direction_col

                    # Check if the new position is within the maze boundaries
                    # and if it has not been visited (maze cell is '.')
                    if 0 <= new_row < rows and 0 <= new_col < cols and maze[new_row][new_col] == '.':
                        # Check if the new position is on the edge of the maze, which means an exit is found
                        if new_row == 0 or new_row == rows - 1 or new_col == 0 or new_col == cols - 1:
                            return steps

                        # Otherwise, add the position to the queue and mark as visited
                        queue.append((new_row, new_col))
                        maze[new_row][new_col] = '+'

        # If we have not found an exit, return -1 indicating failure to find an exit
        return -1

```

Java Solution

```java
class Solution {
    public int nearestExit(char[][] maze, int[] entrance) {
        // Maze dimensions
        int rowCount = maze.length;
        int colCount = maze[0].length;

        // Queue for BFS
        Queue<int[]> queue = new LinkedList<>();
        queue.offer(entrance);

        // Mark the entrance as visited
        maze[entrance[0]][entrance[1]] = '+';

        // Step count for nearest exit
        int steps = 0;

        // Directions for exploring neighbors (up, right, down, left)
        int[] directions = {-1, 0, 1, 0, -1};

        // Begin BFS
        while (!queue.isEmpty()) {
            steps++; // Increment steps at each level

            for (int count = queue.size(); count > 0; count--) {
                int[] currentPos = queue.poll(); // Poll the current position from the queue

                // Iterate through all possible directions
                for (int l = 0; l < 4; l++) {
                    int nextRow = currentPos[0] + directions[l];
                    int nextCol = currentPos[1] + directions[l + 1];

                    // Check if the next position is within bounds and not a wall
                    if (nextRow >= 0 && nextRow < rowCount && nextCol >= 0 && nextCol < colCount && maze[nextRow][nextCol] == '.')
                        // Check if the next position is at the border, thus an exit
                        if (nextRow == 0 || nextRow == rowCount - 1 || nextCol == 0 || nextCol == colCount - 1) {
                            return steps; // Return the number of steps to reach this exit
                        }
                        // Mark the position as visited
                        queue.offer(new int[] {nextRow, nextCol});
                        maze[nextRow][nextCol] = '+';
                    }
                }
            }
        }
        // If no exit was found, return -1
        return -1;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <queue>
using namespace std;

class Solution {
public:
    int nearestExit(vector<vector<char>>& maze, vector<int>& entrance) {
        // Initialize the maze dimensions
        int rows = maze.size(), cols = maze[0].size();

        // Queue to store the path in (x, y) coordinates
        queue<vector<int>> queue{{entrance}};

        // Mark the entrance as visited
        maze[entrance[0]][entrance[1]] = '+';

        // Initialize the step counter
        int steps = 0;

        // Possible directions to move: up, right, down, left
        vector<int> directions = {-1, 0, 1, 0, -1};

        // Perform breadth-first search to find the nearest exit
        while (!queue.empty()) {
            // Increment the step counter at the start of each level of BFS
            ++steps;

            // Process all nodes on the current level
            for (int count = queue.size(); count > 0; --count) {
                // Get the current position
                auto position = queue.front();
                queue.pop();

                // Explore all possible directions from the current position
                for (int i = 0; i < 4; ++i) {
                    int x = position[0] + directions[i], y = position[1] + directions[i + 1];

                    // Check if the new position is within bounds and not visited
                    if (x >= 0 && x < rows && y >= 0 && y < cols && maze[x][y] == '.') {
                        // Check if the new position is an exit
                        if (x == 0 || x == rows - 1 || y == 0 || y == cols - 1) return steps;

                        // Add the new position to the queue and mark as visited
                        queue.push({x, y});
                        maze[x][y] = '+';
                    }
                }
            }
        }

        // Return -1 if no exit is found
        return -1;
    }
};

```

Typescript Solution

```typescript
interface Coordinates {
  x: number;
  y: number;
}

function nearestExit(maze: string[][], entrance: [number, number]): number {
  // Initialize the maze dimensions
  const rows = maze.length;
  const cols = maze[0].length;

  // Initialize a queue with the starting point containing x and y coordinates based on the entrance
  const queue: Coordinates[] = [{ x: entrance[0], y: entrance[1] }];

  // Mark the entrance as visited
  maze[entrance[0]][entrance[1]] = '+';

  // Initialize the step counter
  let steps = 0;

  // Possible directions to move: up, right, down, left (represented by x, y deltas)
  const directions = [{ x: -1, y: 0 }, { x: 0, y: 1 }, { x: 1, y: 0 }, { x: 0, y: -1 }];

  // Perform breadth-first search to find the nearest exit
  while (queue.length > 0) {
    // Increment the step counter at the start of each level of BFS
    steps++;

    // Process all nodes on the current level
    for (let count = queue.length; count > 0; count--) {
      // Get the current position by dequeueing from the front of the array
      const position = queue.shift();

      // Explore all possible directions from the current position
      for (const direction of directions) {
        const newX = position.x + direction.x;
        const newY = position.y + direction.y;

        // Check if the new position is within bounds and not visited
        if (newX >= 0 && newX < rows && newY >= 0 && newY < cols && maze[newX][newY] === '.') {
          // Check if the new position is an exit
          if (newX === 0 || newX === rows - 1 || newY === 0 || newY === cols - 1) {
            return steps;
          }

          // Add the new position to the queue and mark as visited
          queue.push({ x: newX, y: newY });
          maze[newX][newY] = '+';
        }
      }
    }
  }

  // Return -1 if no exit is found
  return -1;
}

```

Time and Space Complexity

The time complexity of the given code is `O(M*N)`, where `M` is the number of rows and `N` is the number of columns in the maze. This complexity arises because, in the worst-case scenario, the algorithm needs to traverse all the cells in the maze before finding an exit or determining that no exit is reachable. Traversal is done via breadth-first search (BFS), and each cell is visited at most once, as visited cells are marked with '+' to prevent revisiting.

The space complexity is also `O(M*N)` due to the same reasoning. The space is used to store the queue `q`, which, in the worst case, could contain all the cells as we explore the maze. Furthermore, we modify the maze to mark visited positions, which also takes `O(M*N)` space; however, this does not add to the complexity as it is the same maze that is passed as input and not additional space being used.