490. The Maze

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

Problem Description

In this problem, we're given a 2D array that represents a maze, where `0`s represent empty spaces and `1`s represent walls. The maze is surrounded by walls on all borders. A ball can roll through the maze, but it only stops moving when it hits a wall. Once it's stopped, it can then be rolled again in any of the four cardinal directions (up, down, left, right). The goal is to determine if there is a way to move the ball from its start position to a specified destination by only rolling it in such a way that it stops at the destination. The start position and destination are given as [row, column] coordinates. The problem asks us to return `true` if the ball can reach the destination and stop there, and `false` otherwise.

Intuition

The solution to this problem involves exploring the maze to find if there is a path to the destination. Exploring a maze is a classic problem that is usually solved using graph traversal algorithms like Depth-First Search (DFS) or Breadth-First Search (BFS). The intuition behind using BFS in this particular problem is that we want to explore all possible paths where the ball can roll until it hits a wall.

The main idea is to simulate the rolling of the ball. We start at the initial position and roll the ball in all possible directions until it hits a wall. Once it comes to a stop, we check if the current stopping point is the destination. If it's not, we treat this stopping point as a new start point and continue to roll the ball in all four directions. We repeat this process until we either reach the destination or all possible paths have been explored without reaching it.

We need to keep track of the positions we have already visited to prevent infinite loops. This is because without tracking, we might keep revisiting the same positions over and over again. For this purpose, we use a set to store visited positions.

By using BFS with a queue and a visited set, we can systematically explore the maze and check if a path to the destination exists.

Solution Approach

The implementation of the aforementioned BFS approach is straightforward. To explain how the solution works, let's break it down step by step:

1. **Queue Initialization**: We initialize a queue `q` for the BFS with the starting position of the ball. This queue will be used to store the positions from where we need to continue exploring further.
2. **Visited Set**: To keep track of the positions we have visited so far to avoid re-processing, we initialize a set `vis`.
3. **BFS Loop**: We enter a loop to process each element in the queue until it's empty, which means we've exhausted all possible movements. For each position taken from the queue: a. We calculate potential new positions by moving the ball in all four possible directions until it hits a wall. b. For each of the four directions, we ensure that the movement is within the bounds of the maze and we are moving into empty space (`0`). c. As soon as the ball stops due to hitting a wall, we check if the stopped position is the destination. If it is, we've found a path, and we return `true`. d. If the new stopped position is not the destination and hasn't been visited yet, we add it to the visited set and the queue for further exploration.
4. **Directions Representation**: The directions in which the ball can move are represented by the vectors `[[0, -1], [0, 1], [-1, 0], [1, 0]]`, which correspond to left, right, up, and down movements, respectively.
5. **Exploring Paths**: For each position discovered, the loop continues exploring further by seeing where the ball can go next from these new positions, until it either reaches the destination or there are no more positions to explore (the queue becomes empty).
6. **Return False**: If the queue is exhausted without finding the destination, the function returns `false`, signifying that there is no path that leads to the destination.

The choice of BFS in this case is strategic, as it allows us to explore the maze level by level, considering the shortest paths first and ensuring that once we reach the destination, we've found the optimal solution (should one exist).

Here's the implementation as provided in the reference document:

```python
class Solution:
    def hasPath(
        self, maze: List[List[int]], start: List[int], destination: List[int]
    ) -> bool:
        m, n = len(maze), len(maze[0])
        q = deque([start])
        rs, cs = start
        vis = {(rs, cs)}
        while q:
            i, j = q.popleft()
            for a, b in [[0, -1], [0, 1], [-1, 0], [1, 0]]:
                x, y = i, j
                while 0 <= x + a < m and 0 <= y + b < n and maze[x + a][y + b] == 0:
                    x, y = x + a, y + b
                if [x, y] == destination:
                    return True
                if (x, y) not in vis:
                    vis.add((x, y))
                    q.append((x, y))
        return False
```

This code clearly follows the BFS paradigm, and it allows us to see a practical implementation of the intuition discussed earlier.

Example Walkthrough

Let's take an example maze and walk through the solution using the BFS approach detailed in the content above.

Suppose we have the following 2D maze where `S` is the start position, `D` is the destination, `0` represents open space, and `1` represents a wall:

```
1 1 1 1 1
1 0 0 0 1
1 0 1 0 1
1 S 1 D 1
1 1 1 1 1
```

The start position is marked as `[3, 1]` and the destination is `[3, 3]`. We want to find if there's a path for the ball to roll from `S` to `D`.

1. **Queue Initialization**: Initialize the queue with the start position `(3, 1)`.
2. **Visited Set**: Initialize the visited set and add the start position to it.
3. **BFS Loop**: Now, we start processing the position `(3, 1)` and we need to explore all four directions. We'll simulate the rolling of the ball until it hits a wall:
    - Rolling left is not an option since `(3, 0)` is a wall.
    - Rolling right ends up at `(3, 2)`, just before the wall.
    - Rolling up isn't possible due to hitting a wall immediately.
    - Rolling down hits a wall immediately.
4. We then mark `(3, 2)` as visited and add it to the queue if it's not already visited. In this case, it's a new position so we add it.
5. The queue now has `(3, 2)`, and we pick this and repeat the process. Rolling right from `(3, 2)` makes the ball stop at `(3, 3)`, which is our destination `D`.
6. Once we hit the destination, we return `True`, indicating that there is indeed a path from `S` to `D`.

If we continue the loop without finding the destination, we will eventually run out of positions to check, which would compel us to return `False`, indicating there's no such path. However, in this example, since we did reach the destination, we successfully exit the loop with the correct outcome.

Python Solution

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

class Solution:
    def hasPath(self, maze: List[List[int]], start: List[int], destination: List[int]) -> bool:
        # Get the dimensions of the maze.
        rows, cols = len(maze), len(maze[0])

        # Initialize a queue to perform BFS and add the starting point.
        queue = deque([start])

        # Set to keep track of visited positions to avoid loops.
        visited = set()
        visited.add(tuple(start))

        # Perform BFS from the start position.
        while queue:
            current_position = queue.popleft()
            row, col = current_position

            # Explore all possible directions: left, right, up, down.
            for direction_row, direction_col in [[0, -1], [0, 1], [-1, 0], [1, 0]]:
                new_row, new_col = row, col

                # Move in the current direction as far as possible until hitting a wall.
                while 0 <= new_row + direction_row < rows and 0 <= new_col + direction_col < cols \
                        and maze[new_row + direction_row][new_col + direction_col] == 0:
                    new_row += direction_row
                    new_col += direction_col

                # If the destination is reached, return True.
                if [new_row, new_col] == destination:
                    return True

                # If the new position has not been visited, mark as visited and add to queue.
                if (new_row, new_col) not in visited:
                    visited.add((new_row, new_col))
                    queue.append((new_row, new_col))

        # If no path is found after exploring all possibilities, return False.
        return False
```

Java Solution

```java
class Solution {
    public boolean hasPath(int[][] maze, int[] start, int[] destination) {
        // Dimensions of the maze
        int numRows = maze.length;
        int numCols = maze[0].length;

        // Visited positions marker
        boolean[][] visited = new boolean[numRows][numCols];

        // Mark the start position as visited
        visited[start[0]][start[1]] = true;

        // Initialize a queue for BFS
        Deque<int[]> queue = new LinkedList<>();
        queue.offer(start);

        // Direction vectors: up, right, down, left
        int[] directions = {-1, 0, 1, 0, -1};

        // Perform BFS to find the path
        while (!queue.isEmpty()) {
            int[] position = queue.poll();
            int currentRow = position[0], currentCol = position[1];

            // Traverse in all four directions
            for (int k = 0; k < 4; ++k) {
                int nextRow = currentRow, nextCol = currentCol;
                // Direction modifiers for both row and column
                int rowInc = directions[k], colInc = directions[k + 1];

                // Roll the ball until a wall is hit
                while (
                    nextRow + rowInc >= 0 && nextRow + rowInc < numRows &&
                    nextCol + colInc >= 0 && nextCol + colInc < numCols &&
                    maze[nextRow + rowInc][nextCol + colInc] == 0
                ) {
                    nextRow += rowInc;
                    nextCol += colInc;
                }

                // If the destination is found, return true
                if (nextRow == destination[0] && nextCol == destination[1]) {
                    return true;
                }

                // If the new position is not visited, mark it visited and add to queue
                if (!visited[nextRow][nextCol]) {
                    visited[nextRow][nextCol] = true;
                    queue.offer(new int[] {nextRow, nextCol});
                }
            }
        }

        // Return false if the destination is not reachable
        return false;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to check if there is a path in the maze from start to destination.
    bool hasPath(vector<vector<int>>& maze, vector<int>& start, vector<int>& destination) {
        int rows = maze.size();    // Number of rows in the maze.
        int cols = maze[0].size(); // Number of columns in the maze.

        // Queue to perform BFS with the initial position being the start.
        queue<vector<int>> queue;
        queue.push(start);

        // Visited array to keep track of visited nodes.
        vector<vector<bool>> visited(rows, vector<bool>(cols, false));
        visited[start[0]][start[1]] = true;

        // 4 possible directions to move: up, right, down, left.
        vector<int> directions = {-1, 0, 1, 0, -1};

        // Start BFS.
        while (!queue.empty()) {
            vector<int> position = queue.front();
            queue.pop();
            int currentRow = position[0], currentCol = position[1];

            // Explore all possible 4 directions.
            for (int k = 0; k < 4; ++k) {
                int newRow = currentRow, newCol = currentCol;
                int rowDir = directions[k], colDir = directions[k + 1];

                // Roll the ball until it hits a wall.
                while (newRow + rowDir >= 0 && newRow + rowDir < rows &&
                       newCol + colDir >= 0 && newCol + colDir < cols &&
                       maze[newRow + rowDir][newCol + colDir] == 0) {
                    newRow += rowDir;
                    newCol += colDir;
                }

                // Check if the new position is the destination.
                if (newRow == destination[0] && newCol == destination[1]) return true;

                // If the new position has not been visited, mark as visited and enqueue it.
                if (!visited[newRow][newCol]) {
                    visited[newRow][newCol] = true;
                    queue.push({newRow, newCol});
                }
            }
        }

        // Return false if destination is not reachable.
        return false;
    }
};
```

Typescript Solution

```typescript
// Represents the maze as a 2D array of numbers, 1 for walls and 0 for paths.
type Maze = number[][];

// Represents a position in the maze with format [row, column].
type Position = [number, number];

// Function signature to check if there is a path in the maze from start to destination.
function hasPath(maze: Maze, start: Position, destination: Position): boolean {
    const rows = maze.length; // Number of rows in the maze.
    const cols = maze[0].length; // Number of columns in the maze.

    // Queue to perform BFS, initialized with the starting position.
    const queue: Position[] = [start];

    // Visited matrix to keep track of visited positions.
    const visited: boolean[][] = Array.from({ length: rows }, () => Array(cols).fill(false));
    visited[start[0]][start[1]] = true;

    // Possible directions to move: left, down, right, and up.
    const directions: number[] = [-1, 0, 1, 0, -1];

    // Start BFS exploration.
    while (queue.length > 0) {
        const [currentRow, currentCol] = queue.shift()!; // Assume queue is not empty and position is defined.

        // Explore in all four directions.
        for (let k = 0; k < 4; ++k) {
            let newRow = currentRow, newCol = currentCol;
            const rowDir = directions[k], colDir = directions[k + 1];

            // Roll the ball until it hits a wall or the edge of the maze.
            while (
                newRow + rowDir >= 0 && newRow + rowDir < rows &&
                newCol + colDir >= 0 && newCol + colDir < cols &&
                maze[newRow + rowDir][newCol + colDir] === 0
            ) {
                newRow += rowDir;
                newCol += colDir;
            }

            // If the new position is the destination, return true.
            if (newRow === destination[0] && newCol === destination[1]) {
                return true;
            }

            // If the new position has not been visited, mark as visited and enqueue it.
            if (!visited[newRow][newCol]) {
                visited[newRow][newCol] = true;
                queue.push([newRow, newCol]);
            }
        }
    }

    // If the destination was not reached, return false.
    return false;
}
```

Time and Space Complexity

The given code is an implementation of the Breadth-First Search (BFS) algorithm, which is used to determine whether there is a path from the start to the destination in a given maze using possible movements up, down, left, or right. It stops movement in one direction when it hits a wall or the edge of the maze.

Time Complexity

The time complexity of this function is `O(m * n)`, where `m` is the number of rows, and `n` is the number of columns in the maze. This is because, in the worst case, the algorithm may explore each cell in the maze once. Each cell is added to the queue only once, and each roll in one of four directions continues until it hits a wall, but the rolling action does not increase the total number of cells visited. Each possible space is processed once, so at most, the algorithm will perform `m * n` operations.

Space Complexity

The space complexity of the function is also `O(m * n)`. This is due to the storage required for the visited set `vis`, which in the worst case may contain every cell in the maze. Moreover, the queue `q` may also store a significant number of positions although not more than `m * n`. However since we only store each cell at most once, the space complexity remains `O(m * n)` overall.