1091. Shortest Path in Binary Matrix

Medium | Breadth-First Search | Array | Matrix

Leetcode Link

Problem Description

The problem presents us with an `n x n` binary matrix `grid`, where the objective is to find the shortest path from the top-left corner `(0, 0)` to the bottom-right corner `(n - 1, n - 1)`. The path is considered clear if it consists only of `0`'s and all adjacent cells in the path (including diagonally adjacent cells) are connected, meaning every step in the path can move horizontally, vertically, or diagonally to the next cell as long as it remains within the bounds of the grid and is a `0`. The length of the path is defined by the number of cells that the path visits. If no such path exists, the function should return `-1`.

Intuition

To solve this problem, we can employ a Breadth-First Search (BFS) approach. BFS is ideal for finding the shortest path in an unweighted grid because it explores all neighboring cells at the current depth level before moving on to cells at the next level of depth. Here's how we get there:

1. **Starting Point**: First, we check if the starting cell `(0, 0)` is a `1` (which would block the path). If it is, we immediately return `-1` since we cannot start the path. If it's a `0`, we can proceed by marking it as visited (`1` in this implementation) to prevent backtracking and initialize our queue with this position.

2. **The Queue**: We use a queue to manage the cells to visit next. It initially contains just the starting cell.

3. **Visiting Cells**: In each step of the BFS, we pop a cell from the queue, process it, and add all its unvisited `0`-neighbors to the queue.

4. **Movement**: We move to adjacent cells in all 8 possible directions. Since the problem specifies 8-directional connectivity, we have to check all cells around the current cell (horizontally, vertically, and diagonally).

5. **Goal Check**: Each time we pop a cell from the queue, we check if it's the bottom-right cell `(n - 1, n - 1)`. If so, we have reached the destination, and we can return the current path length.

6. **Incrementing Path Length**: Every time we've checked all neighbors for the current level (cells with the same path length), we increment the path length before moving on to the next level.

7. **Base Case for No Path**: If we exhaust the queue without reaching the destination, we return `-1`, indicating there is no possible path.

The BFS guarantees that the first time we reach the bottom-right cell, that path is the shortest because we have explored in a way that checks all possible paths of incrementally longer lengths. The grid modification (`grid[x][y] = 1`) acts as a visited marker to avoid revisiting cells and potentially creating loops.

Solution Approach

The solution approach employs Breadth-First Search (BFS) to systematically search for the shortest path from the start to the end of the binary matrix. The BFS algorithm is ideal for such a problem since it exhaustively explores all neighbors of a given cell before moving further away, thereby ensuring the shortest path is found if it exists. Given below is a step-by-step walkthrough of the algorithm:

1. **Validate Start**: Before starting the search, we check if the starting cell (top-left corner) is `0`. If it's `1` (blocking the path), we return `-1`.

2. **Initial Setup**: We initialise our queue, `q`, with a tuple representing the starting cell coordinates `(0, 0)` and set the value of the starting cell to `1` to mark it as visited. The `ans` variable is initialized to `1`, indicating the current path length we're at (starting with the first cell).

3. **Queue Processing**: We create a loop that runs as long as the queue is not empty. This queue will store the cells to explore at each level of depth.

4. **Current Level Exploration**: Inside the loop, we have an inner loop `for _ in range(len(q)):`. This ensures that we only process cells that are at the current level of depth, thus maintaining the BFS property.

5. **Neighbor Exploration**: We pop the front cell from the queue and check all possible 8 directions around this cell. For each of those directions, we check whether the cell is within the grid bounds and whether it is unvisited (`0`). If these conditions are met, we mark the cell as visited (`grid[x][y] = 1`) to prevent revisiting it and add its coordinates to the queue for further exploration.

6. **Goal Condition**: If we encounter the bottom-right cell during exploration, we immediately return the current path length (`ans`), as this is the shortest path due to the BFS.

7. **Incrementing Path Length**: Once we have explored all neighbors of the current depth level, we increment `ans` by 1 to account for the next level that we'll start exploring in the next iteration of the outer loop.

8. **Returning -1**: If the loop terminates without finding the bottom-right cell, we return `-1`, signifying it doesn't have an accessible path.

The choice of a queue in BFS is a fundamental part of the algorithm. It ensures that nodes are explored in the order of their proximity to the start node (level by level, not depth by depth like DFS), which is why the path length is incremented once per level, not per node.

By modifying the grid in-place, we can keep track of visited nodes without the need for an additional visited matrix, which also helps in reducing space complexity.

Overall, the algorithm has a time complexity of `O(n^2)` if all cells are visited in the worst case and a space complexity of `O(n^2)` as well, taken up by the queue in the worst case where all cells are added to it.

Example Walkthrough

Let's take a small `3x3` binary matrix `grid` as an example to illustrate the solution approach:

```
grid = [
    [0, 0, 0],
    [0, 1, 0],
    [1, 0, 0]
]
```

We want to find the shortest path from the top-left corner `(0, 0)` to the bottom-right corner `(2, 2)` using the BFS approach. Here's a step-by-step walkthrough of what the algorithm would do:

1. **Validate Start**: We check if the starting cell `grid[0][0]` is `0`. Since it is, we can proceed.

2. **Initial Setup**: We initialize our queue `q` with `((0, 0), 1)`, the starting cell coordinates and the initial path length. The cell `grid[0][0]` is marked as visited by setting it to `1`.

3. **Queue Processing**: We begin the loop since our queue is not empty.

4. **Current Level Exploration**: Our `q` initially contains one element, so we will explore this level with just one loop iteration.

5. **Neighbor Exploration**: We pop `(0, 0)` from the queue. We explore all possible neighbors:
   - The right cell `(0, 1)` is within bounds and is `0`. We mark it visited, and add it to the queue.
   - The bottom cell `(1, 0)` is within bounds and is `0`. We mark it visited, and add it to the queue.
   - The diagonal cell `(1, 1)` is within bounds but is `1`, so we don't add it to the queue.

   Our queue now looks like this (with corresponding path lengths): `[((0, 1), 2), ((1, 0), 2)]`.

6. **Incrementing Path Length**: At this point we complete the first level, so if we didn't find the end cell, we would proceed to the next level and increment `ans`. Since we found multiple neighbors, `ans` would be `2`.

7. **Processing Next Level**: We continue with our BFS loop for the next level.
   - We first process `(0, 1)`. Its right neighbor is outside the grid, bottom neighbor `(1, 1)` is blocked, and the diagonal bottom-right neighbor `(1, 2)` is in bounds and `0`. We add `(1, 2)` to the queue and mark it as visited.
   - Next, we process `(1, 0)`. Its neighbors to the right `(1, 1)` and bottom `(2, 0)` are blocked, but the diagonal `(2, 1)` is in bounds and `0`. We mark it and add it to the queue.

   Our queue and corresponding path lengths now look like this: `[ (1, 2), 3), ((2, 1), 3)]`.

8. **Reaching Goal**: Continuing the loop, we then process `(1, 2)`; its diagonal neighbor is the end cell `(2, 2)`, and since it is `0`, it means we have reached our destination. We return the current path length `ans`, which is now `3`.

Since we've reached the goal on this level, we don't need to process any further levels. The shortest path length from the top-left corner to the bottom-right corner is `3`.

If at any point we had exhausted the queue without reaching the bottom-right corner, we would return `-1`. In this example, that is not the case, as we have successfully found a path.

Python Solution

```python
from collections import deque

class Solution:
    def shortestPathBinaryMatrix(self, grid: List[List[int]]) -> int:
        # Check if the starting cell is blocked
        if grid[0][0] != 0:
            return -1

        # Initialize the size of the grid
        n = len(grid)
        # Set starting point as visited by marking it with a 1 (path length from the origin)
        grid[0][0] = 1
        # Initialize a queue with the starting coordinate
        queue = deque([(0, 0)])
        # Initialize path length
        path_length = 1

        # Process nodes until the queue is empty
        while queue:
            # Loop through all nodes at the current level of breadth
            for _ in range(len(queue)):
                i, j = queue.popleft()  # Get next node coordinates

                # Check if we've reached the target cell
                if i == j == n - 1:
                    return path_length

                # Check all 8 adjacent cells
                for x in range(i - 1, i + 2):
                    for y in range(j - 1, j + 2):
                        # Boundary check and cell is not blocked
                        if 0 <= x < n and 0 <= y < n and grid[x][y] == 0:
                            # Mark cell as visited and add its coordinates to the queue
                            grid[x][y] = 1
                            queue.append((x, y))
            # Increment path length at the conclusion of the level
            path_length += 1

        # If we exit the loop without returning, no path has been found
        return -1

```

Java Solution

```java
class Solution {
    // Method to find the shortest path in a binary matrix from top-left to bottom-right
    public int shortestPathBinaryMatrix(int[][] grid) {
        // If the starting cell is blocked, return -1
        if (grid[0][0] == 1) {
            return -1;
        }

        // Get the size of the grid
        int n = grid.length;

        // Mark the starting cell as visited by setting it to 1
        grid[0][0] = 1;

        // Initialize a queue to hold the cells to be visited
        Deque<int[]> queue = new ArrayDeque<>();

        // Start from the top-left corner (0, 0)
        queue.offer(new int[] {0, 0});

        // Variable to keep track of the number of steps taken
        for (int steps = 1; !queue.isEmpty(); ++steps) {
            // Process cells level by level
            for (int k = queue.size(); k > 0; --k) {
                // Poll the current cell from the queue
                int[] cell = queue.poll();
                int i = cell[0], j = cell[1];

                // If we have reached the bottom-right corner, return the number of steps
                if (i == n - 1 && j == n - 1) {
                    return steps;
                }

                // Explore all 8 directions from the current cell
                for (int x = i - 1; x <= i + 1; ++x) {
                    for (int y = j - 1; y <= j + 1; ++y) {
                        // Check for valid cell coordinates and if the cell is not blocked
                        if (x >= 0 && x < n && y >= 0 && y < n && grid[x][y] == 0) {
                            // Mark the cell as visited
                            grid[x][y] = 1;

                            // Add the cell to the queue to explore its neighbors later
                            queue.offer(new int[] {x, y});
                        }
                    }
                }
            }
        }

        // If the goal was not reached, return -1
        return -1;
    }
}

```

C++ Solution

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

class Solution {
public:
    int shortestPathBinaryMatrix(vector<vector<int>>& grid) {
        // If the starting cell is blocked, there is no path.
        if (grid[0][0] != 0) {
            return -1;
        }

        int n = grid.size(); // Dimension of the grid
        grid[0][0] = 1; // Mark the starting cell as visited by setting it to 1
        queue<pair<int, int>> q; // Define a queue to store the cells to visit
        q.emplace(0, 0); // Start with the top-left corner of the grid

        // Loop until there are no more cells to visit
        for (int pathLength = 1; !q.empty(); ++pathLength) {
            for (int k = q.size(); k > 0; --k) {
                auto [row, col] = q.front(); // Get the current cell's position
                q.pop(); // Remove the current cell from the queue

                // If the current cell is the bottom-right corner, the path is found
                if (row == n - 1 && col == n - 1) {
                    return pathLength;
                }

                // Iterate through all the neighbors of the current cell
                for (int x = row - 1; x <= row + 1; ++x) {
                    for (int y = col - 1; y <= col + 1; ++y) {
                        // Check if the neighbor is within the grid and is not blocked
                        if (x >= 0 && x < n && y >= 0 && y < n && grid[x][y] == 0) {
                            grid[x][y] = 1; // Mark the neighbor as visited
                            q.emplace(x, y); // Add the neighbor to the queue
                        }
                    }
                }
            }
        }

        // If there is no path, return -1
        return -1;
    }
};

```

Typescript Solution

```typescript
function shortestPathBinaryMatrix(grid: number[][]): number {
    // Early exit if the starting cell (0, 0) is blocked.
    if (grid[0][0] === 1) {
        return -1;
    }

    const gridSize = grid.length;
    // Occupying the starting point.
    grid[0][0] = 1;

    // Queue for the BFS, starting with position (0, 0).
    let queue: number[][] = [[0, 0]];

    // BFS loop. 'steps' counts the number of steps taken.
    for (let steps = 1; queue.length > 0; ++steps) {
        // 'nextQueue' will store the positions to visit in the next loop iteration.
        const nextQueue: number[][] = [];

        // Process each cell in the current queue.
        for (const [row, col] of queue) {
            // Check if the bottom-right corner is reached.
            if (row === gridSize - 1 && col === gridSize - 1) {
                return steps;
            }
            // Explore all 8 directions around the current cell.
            for (let x = row - 1; x <= row + 1; ++x) {
                for (let y = col - 1; y <= col + 1; ++y) {
                    // Check if the new position is valid and not blocked.
                    if (x >= 0 && x < gridSize && y >= 0 && y < gridSize && grid[x][y] === 0) {
                        // Mark the cell as visited by setting it to 1.
                        grid[x][y] = 1;
                        // Add the position to the 'nextQueue'.
                        nextQueue.push([x, y]);
                    }
                }
            }
        }

        // Update the queue for the next iteration of BFS.
        queue = nextQueue;
    }

    // If the bottom-right corner was never reached, return -1.
    return -1;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the algorithm is `O(N^2)`. Here's why:

- The algorithm employs Breadth-First Search (BFS) which can visit each cell at most once. Since it accounts for 8 possible directions a cell can have, the loop check all adjacent 8 cells.
- In the worst case, we have to visit all cells in an `N x N` grid. Each cell is visited once, hence the complexity is proportional to the total number of cells, which is `N^2`.

Space Complexity

The space complexity of the algorithm is `O(N^2)` as well.

- The queue `q` can potentially store all of the cells in case of a sparse grid (consisting mostly of 0s). Hence, in the worst case, queue space can go up to `N^2`.
- The `grid` itself is modified in place, hence no extra space is used other than the input size, which does not count towards space complexity in this context.
- No additional significant space is used, as other variables have constant size.