1730. Shortest Path to Get Food

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

Problem Description

In this problem, you're placed in a grid that represents a map, where your goal is to find the food as quickly as possible. The map includes your current location, marked with a '*', food cells marked with '#', open spaces marked with 'O', and obstacles marked with 'X'. You can move in the four cardinal directions (north, east, south, west) unless an obstacle blocks the way. Your task is to find the length of the shortest path to any of the food cells. Should there be no way to reach any food cell, you must return `-1`.

Key points of the problem:

- The grid is a two-dimensional array made up of characters that represent different types of cells.
- You can only move to adjacent cells (up, down, left, or right), not diagonally.
- You can't pass through 'X' cells, as these are obstacles.
- You need to return the shortest number of steps to reach any food cell, not all food cells.

Intuition

The solution to this problem lies in exploring the grid efficiently to find the shortest path to the nearest food cell. A common approach to solving pathfinding problems is to use a breadth-first search (BFS) algorithm. BFS is ideal here because it explores the nearest neighbors first and gradually goes farther away from the start. This feature guarantees that we find the shortest path to the nearest food source.

Here are the steps of the intuition behind the BFS algorithm used:

1. Locate the starting point where the character '*' is located.
2. Start the BFS from the starting point, adding the initial position to the queue.
3. Explore all neighboring cells in the order of north, east, south, and west.
4. For every valid neighboring cell:
   - If it's a food cell, return the current step count as we've found the shortest path.
   - If it's a free space, mark it as visited by turning it into an obstacle (to avoid revisiting) and add its position to the queue to explore its neighbors in subsequent steps.
5. The BFS continues until either a food cell is found, or all reachable cells have been explored.
6. If no food cell is found and exploration is over, return `-1` indicating the food is unreachable.

The main idea of this solution is to perform the BFS until the food is found while keeping track of the number of steps taken, ensuring the shortest path is always sought.

Solution Approach

The reference solution approach simply mentions "BFS," indicating that Breadth-First Search is the chosen method for solving the problem. This is the correct approach because BFS ensures we explore paths emanating from the source in an expanding 'wave', where each 'wave' represents an increment in the number of steps required to reach that point. As soon as we encounter a food cell on a wavefront, we can be sure it's the closest, because if there were a closer one, we would have encountered it on an earlier wave.

To implement BFS, we commonly use a queue data structure because it naturally processes elements in a first-in-first-out (FIFO) order, which aligns with the BFS pattern of visiting all neighbours of a node before moving on to the next level of nodes. In Python, this is often implemented with a deque (double-ended queue) from the collections module for efficient appending and popping.

Let's explain the given Python code provided:

1. **Initial Setup**

   ```
   m, n = len(grid), len(grid[0])
   i, j = next((i, j) for i in range(m) for j in range(n) if grid[i][j] == '*')
   q = deque([(i, j)])
   dirs = (-1, 0, 1, 0, -1)
   ```

   We start by determining the grid dimensions. The starting point `'*'` is found using a generator expression that iterates over each cell of the grid. Once found, the location `(i, j)` is added to the queue `q`. We also create a tuple of directions `dirs` that will help us traverse the grid in north, east, south, and west directions.

2. **BFS Loop**

   ```
   ans = 0
   while q:
       ans += 1
       for _ in range(len(q)):
           i, j = q.popleft()
           for a, b in pairwise(dirs):
               x, y = i + a, j + b
               ...
   ```

   A while loop begins the BFS process. The count `ans` will be incremented with each 'wave' of the BFS to reflect the number of steps taken to reach that point. Using `len(q)` ensures that only the cells that were in the queue at the start of this depth level are processed, which maintains the separation between BFS levels.

3. **Neighbor Traversal and Checks**

   ```
               ...
               if 0 <= x < m and 0 <= y < n:
                   if grid[x][y] == '#':
                       return ans
                   if grid[x][y] == 'O':
                       grid[x][y] = 'X'
                       q.append((x, y))
   ```

   For each node `(i, j)` processed at the current BFS level, we look at all the possible neighbours by iterating `pairwise` through the `dirs` tuple. If a neighbour `(x, y)` is within the bounds of the grid and is a food cell, we return `ans` immediately as we've found the shortest path. If it's a free space, we mark it as visited by setting it to `'X'` and add its position to the queue to visit its neighbors at the next level.

4. **Handling No Path**

   ```
   return -1
   ```

   If the BFS completes without finding a food cell (`'#'`), which means the queue `q` becomes empty, we return `-1` signifying there's no path available.

This implementation efficiently finds the shortest path using BFS, with modifications to mark visited nodes and return the path length immediately upon reaching a food cell.

Example Walkthrough

Let's assume we have the following small grid that represents the map:

```
[
  ['*', 'O', 'O', 'X'],
  ['O', 'X', 'O', '#'],
  ['O', 'O', 'X', 'O']
]
```

Here is a step-by-step explanation of the algorithm being applied to this grid:

1. **Initial Setup**: We find the starting position where `'*'` is located at `(0, 0)`. We place this position in our BFS queue. Our `dirs` tuple will help us move through the grid in the cardinal directions.
2. **BFS Loop**: We begin a `while` loop with the BFS process, starting with `ans = 0`.
3. **First Wave**: On the first wave (first iteration of the `while` loop), the algorithm checks the neighbours of `(0, 0)`. The open spaces on the east `('O', (0, 1))` and south `('O', (1, 0))` are added to the BFS queue. The obstacle `('X', (0, 3))` and the out-of-bound north direction are ignored. The open spaces are then marked as visited by changing them to `'X'`.
4. **Increment Step Count**: `ans` is incremented to `1`.
5. **Second Wave**: On the second wave, the algorithm checks for neighbours of `(0, 1)` and `(1, 0)` from the queue. For `(0, 1)`, we check `'(1, 1)'`, `'(0, 2)'`, `'(0, 1)'` and `'(2, 1)'`. The only valid movement here is to `'(0, 2)'` which is an open space. For `(1, 0)`, we can only move to `'(2, 0)'` since the east is blocked and the south and west cells are out of bounds or visited, respectively. We mark the newly found open spaces as visited.
6. **Increment Step Count**: `ans` is incremented to `2`.
7. **Third Wave**: The algorithm now checks the neighbours of `(0, 2)` and `(2, 0)`. From `(0, 2)`, we can only move south to `(1, 2)`. From `(2, 0)`, we can move east to `(2, 1)` and south to `(3, 0)`, but `(3, 0)` is out of bounds, so only `(2, 1)` is considered. These cells are added to the queue and marked as visited.
8. **Increment Step Count**: `ans` is incremented to `3`.
9. **Fourth Wave**: The algorithm checks neighbour `(1, 2)`. Here, we find the food cell marked with `'#'` at `(1, 3)` which is east of our current position. Upon finding the food cell, the BFS algorithm terminates.
10. **Return Shortest Path**: Since the food cell is found on the fourth wave, the shortest path length `3` is returned.

Python Solution

```python
from collections import deque

class Solution:
    def getFood(self, grid: List[List[str]]) -> int:
        # Find the size of the grid.
        rows, cols = len(grid), len(grid[0])

        # Find the starting position (where the person is located, marked with '*').
        start_row, start_col = next((row, col) for row in range(rows) for col in range(cols) if grid[row][col] == '*')

        # Initialize a queue for breadth-first search (BFS) with the starting position.
        queue = deque([(start_row, start_col)])

        # Define possible directions to move: up, right, down, and left.
        directions = ((-1, 0), (0, 1), (1, 0), (0, -1))

        # Steps taken, initially 0 as we start from the person's position.
        steps = 0

        while queue:
            # Increase the steps taken each time we explore a new layer of neighbors.
            steps += 1

            # Explore all positions in the current layer.
            for _ in range(len(queue)):
                cur_row, cur_col = queue.popleft()

                # Explore all adjacent positions.
                for d_row, d_col in directions:
                    new_row, new_col = cur_row + d_row, cur_col + d_col

                    # Check if the new position is within the grid bounds.
                    if 0 <= new_row < rows and 0 <= new_col < cols:
                        # Check if we found food at the new position.
                        if grid[new_row][new_col] == '#':
                            return steps

                        # If the new position is open space, mark as visited and add to queue.
                        if grid[new_row][new_col] == 'O':
                            grid[new_row][new_col] = 'X'  # Mark as visited
                            queue.append((new_row, new_col))

        # If we haven't found food and exhausted all positions, return -1.
        return -1

```

Java Solution

```java
class Solution {
    // Directions represent movement as up, right, down, left (4-connected grid directions)
    private int[] directions = {-1, 0, 1, 0, -1};

    public int getFood(char[][] grid) {
        int rows = grid.length, cols = grid[0].length;
        Deque<int[]> queue = new ArrayDeque<>();
        // Search for the starting point represented by '*' and add it to the queue
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                if (grid[i][j] == '*') {
                    queue.offer(new int[] {i, j});
                    break; // Exit the loop once the starting point is found
                }
            }
        }

        // Initialize number of steps taken to reach the food
        int steps = 0;

        // Perform BFS (Breadth-First Search) to find the shortest path to the food
        while (!queue.isEmpty()) {
            ++steps; // Increment steps at the start of each level of BFS
            for (int size = queue.size(); size > 0; --size) {
                // Poll the current position from the queue
                int[] position = queue.poll();

                // Explore all possible next positions using the predefined directions
                for (int k = 0; k < 4; ++k) {
                    int x = position[0] + directions[k];
                    int y = position[1] + directions[k + 1];
                    // Ensure the next position is within the grid boundaries
                    if (x >= 0 && x < rows && y >= 0 && y < cols) {
                        // Check if the food ('#') is found at the current position
                        if (grid[x][y] == '#') {
                            return steps; // Return the number of steps taken
                        }
                        // Mark visited paths as 'X' and add the new cell to the queue if it's open ('O')
                        if (grid[x][y] == 'O') {
                            grid[x][y] = 'X';
                            queue.offer(new int[] {x, y});
                        }
                    }
                }
            }
        }
        // Return -1 if the food cannot be reached
        return -1;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Since we will be moving in 4 directions, define an array for the movements
    const static inline vector<int> directions = {-1, 0, 1, 0, -1};

    // Method to get the minimum steps to reach food from start (*) in the grid
    int getFood(vector<vector<char>>& grid) {
        int rows = grid.size(), cols = grid[0].size(); // Get the size of the grid
        queue<pair<int, int>> toVisit; // Queue to manage BFS

        // Find the starting position and add it to the queue
        for (int row = 0; row < rows; ++row) {
            for (int col = 0; col < cols; ++col) {
                if (grid[row][col] == '*') {
                    toVisit.emplace(row, col);
                    break; // Start is found, no need to continue
                }
            }
        }

        int steps = 0; // Initialize the step counter

        // Perform BFS until queue is empty
        while (!toVisit.empty()) {
            ++steps; // Increase step counter at each level of BFS
            for (int sz = toVisit.size(); sz > 0; --sz) { // Process all nodes at current level
                auto [currentRow, currentCol] = toVisit.front();
                toVisit.pop(); // Remove the current position from the queue

                for (int k = 0; k < 4; ++k) { // Check all 4 directions
                    int newRow = currentRow + directions[k], newCol = currentCol + directions[k + 1];
                    if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols) { // Check bounds
                        if (grid[newRow][newCol] == '#') {
                            return steps; // Food is found, return answer
                        }
                        if (grid[newRow][newCol] == 'O') { // Check if open space
                            grid[newRow][newCol] = 'X'; // Mark as visited
                            toVisit.emplace(newRow, newCol); // Add new position to visit
                        }
                    }
                }
            }
        }

        return -1; // Return -1 if food cannot be reached
    }
};

```

Typescript Solution

```typescript
/**
 * Find the shortest path from a starting point (marked with an asterisk *) to a point with food (marked with a hash #).
 * The function performs a Breadth-First Search (BFS) on the grid.
 *
 * @param {string[][]} grid The grid represented as a 2D array of characters.
 * @return {number} The minimum number of steps to reach food from the starting point, or -1 if it cannot be reached.
 */
function getFood(grid: string[][]): number {
    const rows = grid.length;  // The number of rows in the grid.
    const cols = grid[0].length;  // The number of columns in the grid.
    // Directions array representing the movement in the adjacent cells (up, right, down, left).
    const directions = [-1, 0, 1, 0, -1];
    // The queue used for BFS, storing the coordinates [row, col] of each point.
    const queue: [number, number][] = [];
    // Finding the starting point (where the person is located, marked with an asterisk *) and adding it to the queue.
    for (let row = 0; row < rows; ++row) {
        for (let col = 0; col < cols; ++col) {
            if (grid[row][col] === '*') {
                queue.push([row, col]);
                // Break out of both loops once the starting point is found.
                break;
            }
        }
        // If the queue is not empty, the starting point was found and already added.
        if (queue.length > 0) {
            break;
        }
    }
    // The number of steps taken (initialized to 0, incremented before processing each level).
    let steps = 0;
    // BFS starting from the initial position.
    while (queue.length > 0) {
        steps++;
        // Iterating through all points at the current BFS level.
        for (let size = queue.length; size > 0; --size) {
            const [currentRow, currentCol] = queue.shift()!;
            // Exploring all four directions from the current cell.
            for (let k = 0; k < 4; ++k) {
                const nextRow = currentRow + directions[k];
                const nextCol = currentCol + directions[k + 1];
                // Check if the next cell is within grid bounds.
                if (nextRow >= 0 && nextRow < rows && nextCol >= 0 && nextCol < cols) {
                    // If the cell contains food, return the number of steps.
                    if (grid[nextRow][nextCol] === '#') {
                        return steps;
                    }
                    // If the cell is open, mark it as visited and add to the queue.
                    if (grid[nextRow][nextCol] === 'O') {
                        grid[nextRow][nextCol] = 'X'; // Marking the cell as visited to avoid revisiting it.
                        queue.push([nextRow, nextCol]);
                    }
                }
            }
        }
    }
    // Return -1 if food is not reachable.
    return -1;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(m * n)`. This complexity arises because we perform a breadth-first search (BFS) starting from the initial point where the person is located (denoted by `*`) and exploring all four directions until we find food denoted by `#` or until all accessible points are visited.

The worst-case scenario is when the person is located at one corner of the grid and food is at the opposite corner, requiring the BFS to visit every `O` cell in the `m * n` grid. Since each cell in the grid is pushed to and popped from the queue at most once, and for each cell, the code checks four directions, the time complexity remains linear with respect to the size of the grid.

Space Complexity

The space complexity of the given code is also `O(m * n)` in the worst case due to the usage of a queue to store the cells during the BFS. In the worst-case scenario, if the grid is filled with 'O', which is explorable, the BFS would have added most or all of the cells into the queue before finding the food or determining that food cannot be reached. Thus, the maximum space used by the queue can be proportional to the total number of cells in the grid.

Note that the space complexity includes the space for the `dirs` list and the modified input grid where 'O's are replaced with 'X's as exploration progresses. However, these are not significant in comparison to the space taken by the queue and hence don't affect the overall space complexity order.