909. Snakes and Ladders

Medium Breadth-First Search Array Matrix

Problem Description

In this problem, we are given a square board of `n x n` size, labeled with consecutive numbers from `1` to `n^2` in a Boustrophedon style, meaning that the numbers start from the bottom left corner and snake back and forth row by row; the direction of numbering alternates with each row. The player begins at square `1` and the goal is to reach square `n^2`. On each turn, the player can move to a square in the range of `1` to `6` squares ahead from their current position, as if rolling a six-sided die.

Some squares contain the start of a snake or ladder, indicated by a non-negative number in `board[r][c]`. If a player lands on such a square, they must follow it to the designated end square. The start and end squares of the board don't contain snakes or ladders. The player only follows a snake or ladder for a single move, not chaining multiple snakes or ladders in a single turn.

The objective is to find the minimum number of moves needed to reach the end of the board. We have to consider the effect of dice rolls, the presence of snakes and ladders which can alter the player's position significantly, and the unique movement pattern due to the Boustrophedon style numbering. It's important to avoid getting stuck in cycles caused by avoid infinite loops (visiting the same squares over and over) and to make sure not to follow the same snake or ladder twice in a single move.

Intuition

To solve this problem, we employ a Breadth-First Search (BFS) algorithm. BFS is a suitable approach here because it is excellent for finding the shortest path in an unweighted graph, which essentially our game board is. Each roll of the die represents a potential transition from one node (square) to another. Moreover, since snakes and ladders can move the player to non-adjacent nodes, BFS can handle these jumps seamlessly while keeping track of the shortest path.

We maintain a queue to represent squares we need to visit (game states are just positions on the board). We also keep track of visited positions to avoid processing the same state multiple times. For each turn, we consider all reachable positions within a roll of a six-sided die and follow snakes and ladders immediately as they only affect the current move. By gradually exploring all possibilities, we ensure we don't miss any potential paths that might lead to the goal in fewer moves, and once we reach `n^2`, we can return the number of moves taken.

The function `get(x)` in the solution converts the label of a square to its coordinates on the board, taking into account the alternating direction of the rows. The use of a set `vis` ensures that each square is visited no more than once, optimizing the search and avoiding cycles. The BFS loop increments a counter `ans` which represents the number of moves taken so far until it either finds the end or exhausts all possible paths.

Solution Approach

The implemented solution approach uses Breadth-First Search (BFS), a popular search algorithm, to efficiently find the shortest path to reach the end of the board from the start. Here's the walkthrough of how the solution functions:

1. **BFS Queue:** We use a deque (a queue that allows inserting and removing from both ends) to store the squares we need to visit. Initially, the queue contains just the first square `1`.
2. **Visited Set:** A set `vis` is used to track the squares we have already visited to prevent revisiting and consequently, getting stuck in loops.
3. **Move Count:** `ans` is a counter that keeps track of the number of moves made so far. It gets incremented after exploring all possible moves within one round of a die roll.
4. **Main Loop:** The BFS is implemented in a `while` loop that continues as long as there are squares to visit in the queue.
5. **Exploring Moves:** For each current position, we look at all possible positions we could land on with a single die roll (six possibilities at most), which are the squares numbered from `curr + 1` to `min(curr + 6, n^2)`. We use the `get(x)` function to calculate the corresponding board coordinates from a square number.
6. **Following Snakes or Ladders:** If the calculated position has a snake or ladder (`board[i][j] != -1`), we move to the destination of that snake or ladder as described by the board (`next = board[i][j]`).
7. **Visiting and Queuing:** If the next square (factoring in any snakes or ladders) has not been visited, we add it to the queue and mark it visited.
8. **Checking for Completion:** If, during the BFS, we reach the last square `n^2`, we immediately return the number of moves made (`ans`) since this would represent the shortest path to the end.
9. **Return Case:** If it is not possible to reach the end - which we know if the queue gets exhausted and the end has not been reached - the function returns `-1`.

The BFS algorithm, coupled with the correct handling of the unique numbering and presence of snakes and ladders, ensures an optimized search for the quickest path to victory.

Example Walkthrough

Let's apply the solution approach to a small example to illustrate how it works.

Example Board

Suppose we have a `3x3` board, which means `n=3` and our board looks like this:

```
+---+---+---+
| 7 | 8 | 9 |
+---+---+---+
| 6 | 5 | 4 |
+---+---+---+
| 1 | -2| 3 |
+---+---+---+
```

The `-2` at position `2` means there is a ladder that takes the player directly from square `2` to square `3`. The goal is to reach square `9` in the minimum number of moves.

Walkthrough Steps

1. Initialize the BFS queue with the starting square (`1`), add `1` to the visited set, and set `ans` (the move counter) to `0`.
2. Begin the BFS loop.
   - Dequeue `1`, the current square.
   - Explore all possible moves (squares `2` through `6`, but since our board is `3x3`, squares `4` through `6` do not exist).
   - Square `2` has a ladder to square `3`, so we move to `3` instead of stopping at `2`.
   - Add `3` to the visited set and enqueue it.
3. Increment `ans` to `1` (we've made one move).
4. Dequeue `3` and explore the next moves (`4` through `9`, but since we can reach at most `9` with a roll of six, we only consider up to `9`).
   - None of the squares from `4` to `9` contain a snake or ladder, so we enqueue squares `4`, `5`, `6`, `7`, and `8` and mark them visited.
   - The end square `9` cannot be reached in this turn.
5. Increment `ans` to `2` (we've made two moves).
6. Dequeue `4`, `5`, `6`, `7`, and `8` in the next iterations. From any of these squares, a roll of one or more will reach the end square `9`.
7. Once `9` is reached, return the current `ans` value, which now is `3` as the minimum number of moves taken to reach the end of the board.

Hence, using this method, we determine that the minimum number of moves needed to reach the end of this `3x3` Boustrophedon style snake and ladder board is `3`.

Python Solution

```python
from collections import deque

class Solution:
    def snakesAndLadders(self, board: List[List[int]]) -> int:
        # Helper function to map the board number to board coordinates (i,j)
        def get_square_coordinates(square_number):
            row = (square_number - 1) // board_size, (square_number - 1) % board_size
            if row[0] % 2 == 1:
                # On odd rows, the counting is from right to left.
                col = board_size - 1 - row[1]
            # Flip board to start from bottom of board
            return board_size - 1 - row[0], col

        board_size = len(board)  # n by n board
        total_squares = board_size * board_size
        visited = {1}  # Keep track of visited squares
        steps = 0  # Counter for number of moves

        while queue:
            # Process all squares at the current depth.
            for _ in range(len(queue)):
                current_square = queue.popleft()

                # Win condition: reached the last square
                if current_square == board_size * board_size:
                    return steps

                # Check all possible next moves by dice roll (1-6)
                for next_square in range(current_square + 1, current_square + 7, board_size * board_size + 1):
                    i, j = get_square_coordinates(next_square)

                    # If there's a ladder or snake, take it.
                    if board[i][j] != -1:
                        next_square = board[i][j]

                    # If next square has not been visited, add it to the queue
                    if next_square not in visited:
                        queue.append(next_square)
                        visited.add(next_square)

            # Increment the number of moves after expanding all possible moves at current depth.
            steps += 1

        # If we have exited the loop without reaching the last square, it's not possible to win.
        return -1
```

Java Solution

```java
class Solution {
    private int boardSize;

    // Method to find the shortest path to reach the last square
    public int snakesAndLadders(int[][] board) {
        boardSize = board.length; // Get the size of the board (n x n)
        Deque<Integer> queue = new ArrayDeque<>(); // A queue to perform BFS
        queue.offer(1); // Start from the first square (1-indexed)
        boolean[] visited = new boolean[boardSize * boardSize + 1]; // Visited array to keep track of visited squares
        visited[1] = true; // Mark the first square as visited
        int moves = 0; // Initialize moves counter

        // Perform BFS to reach the last square
        while (!queue.isEmpty()) {
            for (int i = queue.size(); i > 0; --i) { // Iterate over current level
                int currentPosition = queue.poll(); // Get the current square
                if (currentPosition == boardSize * boardSize) { // Check if reached the end
                    return moves;
                }
                // Explore the next 6 possible moves (dice roll)
                for (int nextPosition = currentPosition + 1; nextPosition <= currentPosition + 6 && nextPosition <= boardSize * boardSize; ++nextPosition) {
                    int[] pos = convertTo2DIndex(nextPosition); // Convert square number to board coordinates
                    int next = board[pos[0]][pos[1]]; // Check for ladder
                    if (next == -1) { // Not a ladder
                        next = nextPosition; // Stay on the same square
                    }
                    if (!visited[next]) { // If not visited, mark as visited and add to the queue
                        visited[next] = true;
                        queue.offer(next);
                    }
                }
            }
            ++moves; // Increment move count after finishing one level in BFS
        }
        return -1; // Return -1 if it's impossible to reach the end
    }

    // Convert the square number to board coordinates (i, j)
    private int[] convertTo2DIndex(int squareNum) {
        int row = (squareNum - 1) / boardSize;
        int col = (squareNum - 1) % boardSize;
        // In every row (from the top), the order is right to left
        if (row % 2 == 1) {
            col = boardSize - 1 - col;
        }
        // Convert row to the actual row in the board from the bottom
        return new int[] {boardSize - 1 - row, col};
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int boardSize; // The size of the board is stored at the class level.

    // The main function to play the game on the given board.
    int snakesAndLadders(vector<vector<int>>& board) {
        boardSize = board.size(); // Initial board size
        queue<int> positionsQueue{{1}}; // Starting positions to explore
        int steps = 0; // Starting position is 0 steps away
        vector<bool> visited(boardSize * boardSize + 1, false); // Visited squares
        visited[1] = true; // Start square is visited
        int moves = 0; // Counter for number of moves

        // BFS to find the shortest path.
        while (!positionsQueue.empty()) {
            for (int i = positionsQueue.size(); i > 0; --i) {
                int currentPosition = positionsQueue.front();
                positionsQueue.pop();

                // Check if we've reached the last square
                if (currentPosition == boardSize * boardSize) return moves;

                // Explore each possibility from the current position
                for (int nextPos = currentPosition + 1; nextPos <= min(currentPosition + 6, boardSize * boardSize); ++nextPos) {
                    auto position = convertTo2DIndex(nextPos); // Convert 1D position to 2D grid indices
                    int target = nextPos;
                    if (board[position[0]][position[1]] != -1) { // Check for snakes or ladders
                        // If there is a ladder or snake, move to its end.
                        target = board[position[0]][position[1]];
                    }
                    // If the target square is not visited, add it to queue.
                    if (!visited[target]) {
                        visited[target] = true;
                        positionsQueue.push(target);
                    }
                }
            }
            ++moves; // Increment move count after each level of BFS.
        }
        return -1; // If we are here, there is no solution.
    }

    // Converts a 1D array index to 2D grid indices considering snaking rows.
    pair<int, int> convertTo2DIndex(int num) {
        int row = (num - 1) / boardSize;
        int col = (num - 1) % boardSize;
        if (row % 2 == 1) { // For odd rows we reverse the column index.
            col = boardSize - 1 - col;
        }
        // Convert row from bottom to top because the last row of the board is board.grid 1D index 1.
        row = boardSize - 1 - row;
        return {row, col};
    }
};
```

Typescript Solution

```typescript
// The size of the board is stored as a global variable.
let boardSize: number;

// Converts a 1D array index to 2D grid indices considering snaking rows.
function convertTo2DIndex(num: number): [number, number] {
    const row = Math.floor((num - 1) / boardSize);
    let col = (num - 1) % boardSize;
    if (row % 2 === 1) {
        // For odd rows (according to zero-index), we reverse the column index.
        col = boardSize - 1 - col;
    }
    // Convert row from bottom to top as the last row of the board corresponds to the first 1D index.
    const rowFromBottom = boardSize - 1 - row;
    return [rowFromBottom, col];
}

// The main function to play the game on the given board.
function snakesAndLadders(board: number[][]): number {
    boardSize = board.length; // Initialize boardSize
    const positionsQueue: number[] = [1]; // Queue to hold positions to explore
    let steps = 0; // Starting position is 0 steps away
    const visited = new Array(boardSize * boardSize + 1).fill(false); // Visited squares
    visited[1] = true; // Start square is visited
    let moves = 0; // Counter for number of moves

    // BFS to find the shortest path.
    while (positionsQueue.length > 0) {
        for (let i = positionsQueue.length; i > 0; --i) {
            const currentPosition = positionsQueue.shift()!;

            // Check if we've reached the last square.
            if (currentPosition === boardSize * boardSize) return moves;

            // Explore each possibility from the current position.
            for (let nextPos = currentPosition + 1; nextPos <= Math.min(currentPosition + 6, boardSize * boardSize); ++nextPos) {
                const position = convertTo2DIndex(nextPos); // Convert 1D position to 2D grid indices.
                let target = nextPos;

                // If there is a ladder or snake, move to its end.
                if (board[position[0]][position[1]] !== -1) {
                    target = board[position[0]][position[1]];
                }

                // If the target square is not visited, add it to the queue.
                if (!visited[target]) {
                    visited[target] = true;
                    positionsQueue.push(target);
                }
            }
        }
        ++moves; // Increment the move count after each level of BFS.
    }
    return -1; // If we are here, there is no solution.
}
```

Time and Space Complexity

The given Python code represents a solution for finding the minimum number of moves to reach the end of a snake and ladder board game using a Breadth-First-Search (BFS) algorithm. To analyze the time and space complexity, I will break down the operations performed by the code.

Time Complexity

The time complexity of the solution is determined by the number of vertices in the graph and the number of edges that are explored in the BFS algorithm. Each cell in the board can be considered a vertex, and the possible moves from each cell form the edges.

- The outer while loop runs as long as there are elements in the queue `q`. In the worst case, every cell could be enqueued once, which corresponds to the number of cells in the board, `n * n`.
- For each element in the queue, the inner for loop considers up to 6 possible moves (representing rolling a die).
- The helper function `get`, which translates a linear board position to a matrix position, is constant time, `O(1)`, for each call.

The complexity for each layer of vertices explored is `O(6)` due to the six possible moves (the die roll), resulting in a total time complexity of `O(n^2 * 6)`. Since constant factors are dropped in Big O notation, the overall time complexity simplifies to `O(n^2)`.

Space Complexity

The space complexity of the solution is determined by the additional space required by data structures that store intermediate results or states during the execution of the algorithm.

- The queue `q` can hold all the vertices in the worst case, so its space complexity is `O(n * n)`, since each cell could be visited once.
- The set `vis` also has a space complexity of `O(n * n)` as it may contain a unique entry for every cell in the worst-case scenario.
- The `get` function uses a constant amount of space.

Considering both the queue and visited set, the space complexity is `O(n^2)`.

Therefore, the overall time complexity is `O(n^2)`, and overall space complexity is `O(n^2)`.