1958. Check if Move is Legal

Medium Array Enumeration Matrix Leetcode Link

Problem Description

In the given problem, you have an 8×8 game board represented by a two-dimensional array `board`, where each cell can be in one of three states: it can either be free (marked with `'.'`), hold a white piece (`'W'`), or hold a black piece (`'B'`). Your task is to determine if a move, defined as changing a free cell to a particular color (either white or black), is legal.

A move is legal if it turns the selected cell into the endpoint of a "good line". A good line must follow these conditions:

1. It consists of at least three cells, including endpoints.
2. The endpoints of the line must be of one color.
3. The cells between the endpoints must all be of the opposite color and there should be no free cells.

There must be a good line in any of the eight possible directions from the selected cell: horizontal, vertical, or diagonal. The task boils down to checking whether making the move creates at least one good line according to the above conditions.

Intuition

Given the rules for a legal move, the logical approach is to start from the cell at `(rMove, cMove)` and look in all possible line directions (8 in total) to check if there is a good line with the move being an endpoint.

The solution has the following key steps:

1. Define the eight possible directions to explore from any given cell (up, down, left, right, and the four diagonals).
2. For each direction, keep moving along that line until you either run off the board or encounter a cell that isn't occupied by the opposite color (could be free or the same color as the one you're playing).
3. Count the number of cells of the opposite color you pass in this process.
4. If you find a cell at the end of the sequence that is of the same color as the one you're playing, and there are more than one opposite color cells in between, the move is legal (i.e., a good line is formed).

The provided code creates a list of directions and iterates over them. For each direction, it increments along that path counting the cells until a stopping condition is met. If the final cell checked matches the player's color and there was more than one opposite color cell in between, the function returns `true`, indicating that the move is legal.

Solution Approach

The implementation of the solution involves systematic exploration in every possible direction from the cell where the move is intended. Here is a breakdown of how the algorithm translates into the solution strategy:

1. **Direction Vectors**: The solution uses a list of tuples called `dirs`, where each tuple represents a direction vector in the two-dimensional space of the board. For example, `(1, 0)` represents moving down, `(0, 1)` represents moving right, `(-1, 0)` and `(0, -1)` represent moving up and left respectively, and the four diagonal directions are represented with combinations of 1 and -1.

2. **Iterating Over Directions**: The solution iterates over these direction vectors using a `for` loop. Inside the loop, it sets up two index variables `i` and `j` to the `rMove` and `cMove` coordinates of the move being queried.

3. **Exploring a Direction**: For each of the 8 directions, the code enters a `while` loop which continues as long as the new indices `i + a` and `j + b` (representing the next cell in the direction `(a, b)`) stay within the bounds of the grid (`0` to `n-1`, as the grid size is `8`).

4. **Counting Opposite Colors**: It increments a counter `t` representing the number of cells traversed. The loop breaks if the next cell is either free or of the same color as the move being played (`color`), since this means a good line cannot be assured in this direction.

5. **Check for Legal Move**: After the loop, if the cell that caused the loop to terminate is of the same color as `color`, and if at least one opposite-colored piece (`t > 1`) was found in the traversed path, then the move is legal. A `true` value is immediately returned.

6. **Result**: If none of the directions leads to a legal move, meaning no good line was formed, the function returns `false` after exiting the for loop.

The solution effectively uses a pattern common in grid-based problems, iterating over a set of fixed directions to explore adjacent cells. By using the direction vectors with a while loop and boundary checks, it manages to traverse the two-dimensional array efficiently.

The data structures used here are basic; a list for directions and simple variables for indices and counters. The algorithm does not require additional data structures like stacks, queues, or maps.

This iterative approach examines each potential line emanating from the cell `(rMove, cMove)`, handling the board as if it were an infinite plane, clipping off paths that go off-grid or do not meet the conditions for a legal move.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the following 8×8 board configuration where 'rMove' is at row 3, and 'cMove' is at column 3 (0-indexed), and we wish to place a black piece (`'B'`):

```
. . . . . . . .
. . . . . . . .
. . W B W . . .
. . W . W . . .
. . W B W . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
```

We want to determine if placing a black piece at (3,3) is a legal move.

Following the solution approach:

1. **Direction Vectors**: We have a list of eight possible directions to check - up, down, left, right, and the four diagonals.

2. **Iterating Over Directions**: We start iterating over these directions. Let's consider checking upwards first (direction vector `(-1, 0)`).

3. **Exploring a Direction**: We move up from our desired move position (3,3) to (2,3) and keep going up as long as we're within the bounds and encountering white pieces (`'W'`).

4. **Counting Opposite Colors**: We find that we indeed encounter white pieces and keep a counter `t`. Since we find white pieces at positions (2,3) and (1,3), `t` is now 2.

5. **Check for Legal Move**: As we move to the next cell upwards (0,3), we find a black piece (`'B'`). Since `t > 1` and the piece found is of the same color as the piece we wish to play, this direction confirms a legal move.

6. **Result**: We return `true` before checking the remaining directions because we've found at least one good line (vertical line from (0,3) through (3,3)), and thus the move at (3,3) is legal.

In this example, only one direction was needed to confirm the legality of the move, but in a complete implementation, all eight directions would be checked to evaluate the move fully. This demonstrates how the algorithm systematically checks each direction to determine whether a legal "good line" is formed.

Python Solution

```python
class Solution:
    def checkMove(
        self, board: List[List[str]], r_move: int, c_move: int, color: str
    ) -> bool:
        # Define all 8 possible directions to move in a 2D grid
        directions = [(1, 0), (0, 1), (-1, 0), (0, -1), (1, 1), (1, -1), (-1, 1), (-1, -1)]

        # The size of the Othello board is 8x8
        board_size = 8

        # Check each direction from the move point
        for delta_row, delta_col in directions:
            # Initialize the current position to the initial move point
            row, col = r_move, c_move
            # Track the number of opponent's pieces between our pieces
            in_between_count = 0

            # Move in the current direction until we're still on the board
            while 0 <= row + delta_row < board_size and 0 <= col + delta_col < board_size:
                # Move to the next cell in the direction
                row, col = row + delta_row, col + delta_col

                # If we find a piece of the same color or an empty space, stop looking in this direction
                if board[row][col] in ['.', color]:
                    break

                # Otherwise, we've found an opponent's piece
                in_between_count += 1

            # If the last piece we found was of the same color and there was at least one piece in between,
            # then the move is legal
            if board[row][col] == color and in_between_count > 1:
                return True

        # If no direction is valid, then the move is not legal
        return False
```

Java Solution

```java
class Solution {
    private static final int[][] DIRECTIONS = { // Directions to check for flips
        {1, 0},   // South
        {0, 1},   // East
        {-1, 0},  // North
        {0, -1},  // West
        {1, 1},   // Southeast
        {1, -1},  // Southwest
        {-1, 1},  // Northeast
        {-1, -1}  // Northwest
    };
    private static final int BOARD_SIZE = 8; // Standard Othello board size

    public boolean checkMove(char[][] board, int rowMove, int columnMove, char color) {
        // Loop through all possible directions
        for (int[] direction : DIRECTIONS) {
            int currentRow = rowMove;
            int currentColumn = columnMove;
            int moveLength = 0; // Length of the potential line of opponent's pieces between our pieces
            int rowDelta = direction[0], columnDelta = direction[1];

            // Keep moving in the direction while the next position is inside the board
            while (0 <= currentRow + rowDelta && currentRow + rowDelta < BOARD_SIZE
                    && 0 <= currentColumn + columnDelta && currentColumn + columnDelta < BOARD_SIZE) {
                moveLength++; // Increase the length of the line
                currentRow += rowDelta;
                currentColumn += columnDelta;

                // If the next position is either empty or contains a piece of the same color, break out of the loop
                if (board[currentRow][currentColumn] == '.' || board[currentRow][currentColumn] == color) {
                    break;
                }
            }

            // Check if the last piece in the direction is the same color and the length of opponent's pieces is more than 1
            if (board[currentRow][currentColumn] == color && moveLength > 1) {
                return true; // The move is valid as it brackets at least one line of opponent pieces
            }
        }
        return false; // If no direction is valid, the move is invalid
    }
}
```

C++ Solution

```cpp
#include <vector>
using std::vector;

class Solution {
public:
    // Directions of 8 possible moves from any position (vertical, horizontal, diagonal)
    vector<vector<int>> directions = {
        {1, 0},  // Down
        {0, 1},  // Right
        {-1, 0}, // Up
        {0, -1}, // Left
        {1, 1},  // Down-right diagonal
        {1, -1}, // Down-left diagonal
        {-1, 1}, // Up-right diagonal
        {-1, -1} // Up-left diagonal
    };
    int boardSize = 8; // Board is 8x8

    // Check if a move is valid by the game rules
    bool checkMove(vector<vector<char>>& board, int rowMove, int colMove, char color) {
        // Iterate over all possible directions
        for (auto& direction : directions) {
            int deltaX = direction[0], deltaY = direction[1];
            int currRow = rowMove, currCol = colMove;
            int tilesToFlip = 0; // Counter for the number of opponent's tiles in the line

            // Move in the direction while staying within the bounds of the board
            while (0 <= currRow + deltaX && currRow + deltaX < boardSize &&
                   0 <= currCol + deltaY && currCol + deltaY < boardSize) {
                // Move to the next tile
                currRow += deltaX;
                currCol += deltaY;

                // If the tile is empty or has the same color, move is not valid in this direction
                if (board[currRow][currCol] == '.' || board[currRow][currCol] == color) break;

                // Increase the count of tiles to flip
                tilesToFlip++;
            }

            // Check if after moving in this direction, we end on our own color and there was at least one tile to flip
            if (board[currRow][currCol] == color && tilesToFlip > 1) {
                return true; // Move is valid in this direction
            }
        }
        return false; // Move is not valid in any direction
    }
};
```

Typescript Solution

```typescript
// Possible directions of moves from any position (vertical, horizontal, diagonal)
const directions: number[][] = [
  [1, 0],   // Down
  [0, 1],   // Right
  [-1, 0],  // Up
  [0, -1],  // Left
  [1, 1],   // Down-right diagonal
  [1, -1],  // Down-left diagonal
  [-1, 1],  // Up-right diagonal
  [-1, -1]  // Up-left diagonal
];

const boardSize: number = 8; // The board is an 8x8 grid

/**
 * Check if a move is valid by the game rules.
 *
 * @param {string[][]} board - The game board represented as a two-dimensional array.
 * @param {number} rowMove - The row index of the move to check.
 * @param {number} colMove - The column index of the move to check.
 * @param {string} color - The color of the current player ('B' or 'W').
 * @returns {boolean} True if the move is valid, otherwise false.
 */
function checkMove(board: string[][], rowMove: number, colMove: number, color: string): boolean {
  // Iterate over all possible directions to find if the move is valid
  for (let direction of directions) {
    let deltaX = direction[0], deltaY = direction[1];
    let currRow = rowMove, currCol = colMove;
    let tilesToFlip = 0; // Counter for the number of opponent's tiles in line

    // Continue moving in the direction while within the bounds of the board
    while (0 <= currRow + deltaX && currRow + deltaX < boardSize &&
           0 <= currCol + deltaY && currCol + deltaY < boardSize) {
      // Proceed to the next tile in the direction
      currRow += deltaX;
      currCol += deltaY;

      // If the tile is empty or has the same color, the move is not valid in this direction
      if (board[currRow][currCol] === '.' || board[currRow][currCol] === color) break;

      // Otherwise, increase the count of tiles to flip
      tilesToFlip++;
    }

    // After the loop, if we end on our own color and there was at least one tile to flip, the move is valid
    if (board[currRow] && board[currRow][currCol] === color && tilesToFlip > 0) {
      return true;
    }
  }

  // If the move wasn't found to be valid in any direction, return false
  return false;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by how many times we loop over the different directions from the starting move, as well as how far we can go in each direction. We have 8 possible directions to check, and in the worst-case scenario, we could iterate over all `n` cells in one direction (where `n` is the size of the board's dimension, which is 8 in this case). Therefore, the worst-case time complexity is `O(8n)` which simplifies to `O(n)` because 8 is a constant factor.

In this specific case, since `n` is fixed at 8, we can also argue that the time complexity is `O(1)` since the board size doesn't change and there's a maximum number of steps to be taken.

Space Complexity

The space complexity of the code is `O(1)` since the extra space used does not scale with the size of the input. We have a fixed-size board and the `dirs` array which consists of 8 directions, and a few variables i, j, and t, which all occupy constant space regardless of input size.