2596. Check Knight Tour Configuration

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

Problem Description

In this problem, you are given a chessboard of size `n x n`, where `n` represents both the height and width of the board. The chessboard is represented by an integer matrix `grid`, with each integer being unique and ranging from `0` to `n * n - 1`. The grid shows the order in which a knight has visited each cell on the chessboard, starting from the top-left cell (which should be 0). The knight is supposed to visit every cell exactly once.

The task is to verify if the given grid represents a valid knight's tour. In a knight's tour, the knight starts at the top-left corner and makes moves in an "L" shape pattern: either moving two squares vertically and one square horizontally, or two squares horizontally and one square vertically. Every cell must be visited exactly once in the tour. The problem asks you to return `true` if the grid is a valid tour, or `false` if it is not.

Intuition

To approach this solution, you can simulate the knight's path using the given `grid` and check if each move is valid. Begin by ensuring that the knight starts at the top-left corner, which would be the 0th position in the tour—any other starting position makes the grid invalid immediately.

Next, create an array `pos` that will store the coordinates `(x, y)` for each step the knight takes. It is necessary because the `grid` is a 2D array representing the tour sequence, and you need to derive the actual coordinates visited at each step.

Once you have the coordinates for the entire path the knight took, go through them in pairs and check if the difference between two adjacent coordinates corresponds to a legal knight move. A legal knight move is only valid if the change in x (dx) and change in y (dy) match either `(1, 2)` or `(2, 1)`. If at any point you find that the move does not match these conditions, the tour is invalid, and you return `false`.

If all the moves are legal, once you complete the traversal of the path, you return `true` indicating the grid represents a valid knight's tour.

Solution Approach

The solution begins by creating a list `pos` that will record the coordinates `(i, j)` corresponding to each step of the knight's tour, with `i` being the row index and `j` being the column index in the `grid`. The `grid` holds integers that represent the move sequences, and the `pos` list will be used to store the actual `(x, y)` positions for these moves.

Next, a `for` loop iterates over each cell `(i, j)` of the `grid` to fill out the `pos` list where `pos[grid[i][j]]` will be the tuple `(i, j)`, effectively mapping the move sequence to coordinates. If the knight did not start at the top-left corner (`grid[0][0]` should be 0), then the `grid` is immediately deemed invalid.

After the `pos` list is populated, another `for` loop iterates through `pos` in pairs using the `pairwise` function. For each pair of adjacent positions `(x1, y1)` and `(x2, y2)`, we calculate the absolute differences `dx` as `abs(x1 - x2)` and `dy` as `abs(y1 - y2)`. These differences correspond to the movements made by the knight.

We check if the move is a legal knight's move—a move that results in `dx` being 1 and `dy` being 2 or vice versa. The `ok` variable holds a Boolean representing whether the move is valid `(dx == 1 and dy == 2) or (dx == 2 and dy == 1)`.

If at any point, a pair of positions does not form a legal knight's move, the function returns `false` because this would indicate an invalid knight's tour.

Finally, if all adjacent pairs fulfill the criteria of a knight's valid move, the loop completes without finding any invalid moves, and the function returns `true`, which signifies that the provided `grid` corresponds to a valid configuration of the knight's tour.

This approach relies on the idea of transforming the problem into a simpler data structure (`pos` list) that can be more easily traversed and verified. The `pairwise` iterator is a pattern that helps in checking the adjacent elements without manually handling index increments, making the code more readable and less error-prone.

Example Walkthrough

Let's consider a 5×5 chessboard and a sequence that we need to verify. Let's assume the given `grid` is as follows:

```
0  10  17  6  23
15 5   8   11 18
24 1   14  21 7
9  16  3   20 12
4  13  22  19 2
```

According to our solution approach, we need to first map each move sequence number to its coordinates on the grid.

1. To create the `pos` list, we iterate through the grid:

- `grid[0][0]` is 0, so `pos[0]` becomes `(0, 0)`.
- `grid[0][1]` is 10, so `pos[10]` becomes `(0, 1)`.
- ...
- `grid[4][4]` is 2, so `pos[2]` becomes `(4, 4)`.

Our `pos` list now represents the sequence in which the knight moves on the chessboard.

2. Next, we iterate through the `pos` list to verify if each move is valid.

- For the start: `pos[0]` is `(0, 0)` and `pos[1]` is `(2, 1)`. The move from (0, 0) to (2, 1) has differences `dx = 2` and `dy = 1`, which is a legal knight's move.
- We then look at `pos[1]` to `pos[2]`: moving from `(2, 1)` to `(4, 4)`. The differences `dx = 2` and `dy = 3` do not match the legal moves of a knight. Thus, according to our algorithm, we return `false`.

The `pairwise` iteration helps us quickly check the moves one after the other:

- Current Position = (0, 0), Next Position = (2, 1), Move = Legal
- Current Position = (2, 1), Next Position = (4, 4), Move = Illegal, returns `false`.

We immediately know that the sequence is not a valid knight's tour since the second move is not legal for a knight, so we don't need to check the entire grid.

This example walkthrough effectively illustrates how our algorithm works to verify a knight's tour on the chessboard. By converting move sequences into coordinates and verifying each move one by one, we can efficiently determine the validity of the tour.

Python Solution

```python
from typing import List
from itertools import pairwise  # Python 3.10 introduced pairwise function in itertools

class Solution:
    def check_valid_grid(self, grid: List[List[int]]) -> bool:
        """ Check if the numbers in the grid follow a knight's move pattern

        Args:
        grid: A 2D list representing a square grid

        Returns:
        A boolean value indicating whether the grid is valid under the condition
        that each consecutive number must be a knight's move away from the previous.
        """

        # Ensure the first element is 0 as required by the problem's conditions
        if grid[0][0] != 0:
            return False

        # Get the size of the grid
        size = len(grid)

        # Initialize a list to hold the positions of the numbers in the grid
        positions = [None] * (size * size)

        # Populate the positions list with coordinates of numbers from the grid
        for row_index in range(size):
            for col_index in range(size):
                # Note that the numbers have to be decreased by 1 to become 0-indexed positions
                positions[grid[row_index][col_index] - 1] = (row_index, col_index)

        # Use pairwise from itertools to iterate over consecutive pairs of number positions
        for (row1, col1), (row2, col2) in pairwise(positions):
            # Calculate the absolute deltas between the positions
            delta_row, delta_col = abs(row1 - row2), abs(col1 - col2)

            # Check for valid knight's move: either 1 by 2 steps or 2 by 1 step
            is_valid_knight_move = (delta_row == 1 and delta_col == 2) or (delta_row == 2 and delta_col == 1)
            if not is_valid_knight_move:
                # If any pair of numbers is not separated by a knight's move, the grid is invalid
                return False

        # If all pairs of numbers are separated by a knight's move, the grid is valid
        return True

```

Java Solution

```java
class Solution {
    // Function to check if a given grid represents a valid grid for the given conditions
    public boolean checkValidGrid(int[][] grid) {
        // Check if the first element is 0 as required
        if (grid[0][0] != 0) {
            return false;
        }

        // Calculate the size of the grid
        int gridSize = grid.length;

        // Create a position array to store positions of each number in the grid
        int[][] positions = new int[gridSize * gridSize][2];
        for (int row = 0; row < gridSize; ++row) {
            for (int col = 0; col < gridSize; ++col) {
                // Storing the current number's position
                positions[grid[row][col]] = new int[] {row, col};
            }
        }

        // Loop to check the validity of the grid based on the position of consecutive numbers
        for (int i = 1; i < gridSize * gridSize; ++i) {
            // Get the positions of the current and previous numbers
            int[] previousPosition = positions[i - 1];
            int[] currentPosition = positions[i];

            // Calculate the distance between the current number and the previous number
            int dx = Math.abs(previousPosition[0] - currentPosition[0]);
            int dy = Math.abs(previousPosition[1] - currentPosition[1]);

            // Check if the distance satisfies the condition for a knight's move in chess
            boolean isValidMove = (dx == 1 && dy == 2) || (dx == 2 && dy == 1);
            if (!isValidMove) {
                // If the move is not valid, the grid is not valid
                return false;
            }
        }

        // If all moves are valid, the grid is valid
        return true;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <cmath> // Include <cmath> for abs function

// The Solution class containing a method to check if a grid is valid
class Solution {
public:
    // Method to check if the moves in the grid represent a valid knight's tour
    bool checkValidGrid(std::vector<std::vector<int>>& grid) {
        // The start position must be 0, if not return false
        if (grid[0][0] != 0) {
            return false;
        }

        // Size of the grid
        int gridSize = grid.size();

        // Vector to hold the positions of the numbers in the grid
        std::vector<std::pair<int, int>> numberPositions(gridSize * gridSize);

        // Populate numberPositions with coordinates of each number in the grid
        for (int row = 0; row < gridSize; ++row) {
            for (int col = 0; col < gridSize; ++col) {
                numberPositions[grid[row][col]] = {row, col};
            }
        }

        // Iterate over the numbers in the grid and check if each move is a valid knight move
        for (int number = 1; number < gridSize * gridSize; ++number) {
            // Get the current and previous positions
            auto [previousX, previousY] = numberPositions[number - 1];
            auto [currentX, currentY] = numberPositions[number];

            // Calculate the differences in x and y coordinates
            int deltaX = std::abs(previousX - currentX);
            int deltaY = std::abs(previousY - currentY);

            // Check if the move is a valid knight's move (L shape move)
            bool isValidKnightMove = (deltaX == 1 && deltaY == 2) || (deltaX == 2 && deltaY == 1);

            // If the move isn't valid, return false
            if (!isValidKnightMove) {
                return false;
            }
        }

        // If all moves are valid, the grid represents a valid knight's tour
        return true;
    }
};

```

Typescript Solution

```typescript
function checkValidGrid(grid: number[][]): boolean {
    // Ensure the first element is 0 as expected
    if (grid[0][0] !== 0) {
        return false;
    }

    // Find the size of the grid
    const gridSize = grid.length;

    // Initialize an array to track the positions of the numbers in the grid
    const positions = Array.from({ length: gridSize * gridSize }, () => new Array(2).fill(0));

    // Populate positions array with the coordinates of each number in the grid
    for (let row = 0; row < gridSize; ++row) {
        for (let col = 0; col < gridSize; ++col) {
            positions[grid[row][col]] = [row, col];
        }
    }

    // Validate the positions of all numbers from 1 to n*n - 1
    for (let i = 1; i < gridSize * gridSize; ++i) {
        // Get the position of the current and previous number
        const previousPos = positions[i - 1];
        const currentPos = positions[i];

        // Calculate the absolute differences in x and y directions
        const deltaX = Math.abs(previousPos[0] - currentPos[0]);
        const deltaY = Math.abs(previousPos[1] - currentPos[1]);

        // Check for knight-like move (L-shape): 2 by 1 or 1 by 2 steps
        const isValidMove = (deltaX === 1 && deltaY === 2) || (deltaX === 2 && deltaY === 1);

        // If the move is not valid, return false
        if (!isValidMove) {
            return false;
        }
    }

    // If all positions are valid, return true
    return true;
}

```

Time and Space Complexity

The time complexity of the code is `O(n^2)`. This is because the main computations are in the nested for-loops, which iterate over every cell in the given grid. Since the grid is `n` by `n`, the iterations run for `n^2` times.

The space complexity of the code is `O(n^2)` as well. This stems from creating a list called `pos` of size `n * n`, which is used to store the positions of the integers from the grid in a 1D array. Since the grid stores `n^2` elements, and we're converting it to a 1D array, the space taken by `pos` will also be `n^2`.