688. Knight Probability in Chessboard

Medium Dynamic Programming Leetcode Link

Problem Description

In this problem, you are given an `n x n` chessboard and a knight piece located at a starting cell `(row, column)`. You're asked to calculate the probability that after making exactly `k` moves, the knight remains within the borders of the chessboard, assuming it moves randomly across its legitimate move set.

A knight in chess moves in an L-shape: two squares in a cardinal direction (north, south, east, or west) followed by one square in an orthogonal direction (90-degree turn from the cardinal direction), or vice versa. The challenge here is that at each move, the knight could potentially step off the board, and the task is to compute how likely the knight is to still be on the board after it has made all `k` moves.

The moves the knight makes are direction-agnostic, meaning it doesn't favor any direction and choses its move uniformly at random from the eight possible options at each step, unless the move would take it off the board, in which case the move isn't made.

Intuition

To tackle this problem, we use a technique called Dynamic Programming (DP), which is frequently used to solve problems where you're asked to figure out several outcomes related to different scenarios — in this case, the probability of the knight being on different cells after different numbers of moves.

The key insight in applying Dynamic Programming is the idea that the solution to our problem can be composed of solutions to subproblems. With this problem, we notice that the probability of the knight being on a certain cell after `h` moves depends only on the probabilities of it being on the adjacent cells that can reach the current cell in one move, after `h-1` moves.

Thus, we can build out a three-dimensional table `f`, where `f[h][i][j]` represents the probability of the knight being on cell `(i, j)` after `h` moves. Initially, when `h=0`, the knight hasn't made any moves so it's certainly on its starting cell, hence probabilities across the board are set to 1.

For subsequent moves (`h > 0`), we calculate the probability for each cell based on probabilities of the previous step, averaging over all the knight moves since each move is equally likely. If a move would go off the board, it doesn't contribute to the probabilities because it's an invalid move. This is repeated until we reach `h = k`, at which point `f[k][row][column]` will give us the desired probability.

The intuition behind this approach comes from breaking down the chaotic process of the knight hopping around into manageable pieces where each state (board configuration after a certain number of moves) is the result of smaller, easier-to-calculate probabilities (board configuration after one fewer move).

Solution Approach

The solution employs a Dynamic Programming (DP) algorithm to iteratively compute the probabilities of the knight being on different cells of the chessboard after a certain number of moves. The main data structure used in this solution is a three-dimensional array `f`, where `f[h][i][j]` holds the probability of the knight landing on cell `(i, j)` after making `h` moves.

Let's break down the steps of the algorithm based on the Reference Solution Approach provided:

1. Initialization:
   - We initialize a three-dimensional list `f` with dimensions `(k + 1) x n x n` with zeros.
   - For every cell `(i, j)` on the chessboard, we set `f[0][i][j] = 1` because before making any move, the probability that the knight is on any cell it starts from is 100%.
2. DP Transition:
   - We iterate through each possible number of moves `h` from `1` to `k`.
   - For each number of moves `h`, we iterate over all cells `(i, j)` of the chessboard looking to fill `f[h][i][j]` with the updated probabilities.
   - For each cell `(i, j)`, we iterate over all the possible moves the knight could make from this cell. This results in potentially 8 adjacent cells `(a, b)` from where the knight could have come.
   - For each potential move, we check if the target cell `(a, b)` is within the boundaries of the chessboard.
   - If the move is valid (the cell is on the board), we update `f[h][i][j]` by adding to it the probability `f[h - 1][a][b] / 8`. The division by 8 accounts for the knight's uniform probability of choosing any of its 8 moves.
3. Result:
   - Once we have processed `k` moves, our DP table contains the probabilities of the knight being on each cell for every number of moves up to `k`.
   - We retrieve `f[k][row][column]`, which is the probability of the knight being on its starting cell `(row, column)` after making `k` moves.

The efficiency of this approach comes from the fact that each subproblem (computing `f[h][i][j]`) is solved once and reused to solve other subproblems that depend on it, following the principle of optimal substructure in dynamic programming. This reduces the number of computations from exponentially many naive simulations to a polynomial amount related to the size of the DP table (`k x n x n`).

Example Walkthrough

Let's illustrate the solution approach using an example. Consider a `3x3` chessboard and a knight starting at the cell `(1,1)` (with both dimensions 0-indexed), and we want to compute the probability of the knight being on the chessboard after exactly `2` moves.

1. **Initialization:**
   - We create a three-dimensional list `f` of size `(3 = k + 1) x 3 x 3` and initialize all values to zero.
   - We set `f[0][1][1] = 1` because initially, the knight is at `(1,1)` with a probability of 100%.
2. **First Move Calculations (h = 1):**
   - We look to calculate the probabilities for `h = 1` for all cells.
   - Starting from `(1,1)`, the knight can move to `(0,2)`, `(2,2)`, `(2,0)`, and `(0,0)`, assuming we're considering a 3×3 board where these are still within bounds. Let's account for each valid move from `(1,1)`:
     - For the move to `(0,2)`, we update `f[1][0][2] = f[0][1][1] / 8 = 1/8` because from step 0 to step 1, the knight has a 1/8 chance of making this move (1 out of 8 possible moves).
     - Repeat for the other three valid moves.
   - At the end of this step, the DP list `f` at `h=1` would show `1/8` at the positions `(0,2)`, `(2,2)`, `(2,0)`, and `(0,0)`, and `0` elsewhere.
3. **Second Move Calculations (h = 2):**
   - Now we calculate the probabilities for `h = 2`.
   - For each of the cells that the knight could have moved to in the first move, we consider all possible moves the knight could take from there.
   - Take the cell `(0,2)`:
     - From `(0,2)`, the knight could move to `(2,1)` and `(1,0)`. We update `f[2][2][1] += f[1][0][2] / 8 = 1/8 / 8 = 1/64` and similarly for `(1,0)`.
     - Notice how this time, several moves could contribute to the probability of ending up on the same cell, so we sum up the probabilities from all potential previous positions.
   - We perform this for all cells that were reachable from `(1,1)` in the first step, updating their reachable cells' probabilities.
4. **Result:**
   - Having completed calculations for `h = 2` moves, our DP table `f` now contains the probabilities of the knight being on each cell after `2` moves.
   - We look at `f[2][1][1]` to find the final probability of the knight being on its starting cell after the second move. In this case, our simplified example may result in `f[2][1][1] = 0` as the knight cannot return to `(1,1)` in exactly 2 moves on a `3x3` board.

The result of the DP table would likely show non-zero probabilities around the edges or center of the `3x3` board, depending on the initial position and the number of moves. This example with `k=2` is small-scale; in a real scenario with a large `n x n` board and a higher number of moves, the probabilities would be spread out more and would require careful computation for each step to accumulate the probabilities correctly.

Python Solution

```python
class Solution:
    def knightProbability(self, N: int, K: int, row: int, column: int) -> float:
        # Initialize a 3D DP array with dimensions (K+1) x N x N, where
        # dp[h][i][j] represents the probability of the knight being on board
        # after h moves starting from i, j.
        dp = [[[0] * N for _ in range(N)] for _ in range(K + 1)]

        # After 0 moves, the probability of the knight being on the board
        # starting from any square is 1.
        for r in range(N):
            for c in range(N):
                dp[0][r][c] = 1

        # The 8 possible movements of a knight in chess, represented as (dx, dy).
        movements = [(-2, -1), (-1, -2), (1, -2), (2, -1),
                     (2, 1), (1, 2), (-1, 2), (-2, 1)]

        # Loop through each step from 1 to K.
        for step in range(1, K + 1):
            for r in range(N):
                for c in range(N):
                    # For each cell (r, c), calculate the probability based on the
                    # previous step's positions. We add 1/8th of the probability from
                    # each of the 8 possible positions the knight could have come from.
                    for dr, dc in movements:
                        prev_r, prev_c = r + dr, c + dc
                        # Check if the previous position is on the board.
                        if 0 <= prev_r < N and 0 <= prev_c < N:
                            dp[step][r][c] += dp[step - 1][prev_r][prev_c] / 8.0

        # Return the probability of the knight remaining on the board after K moves.
        return dp[K][row][column]

```

Java Solution

```java
class Solution {
    public double knightProbability(int n, int k, int row, int column) {
        // Create a 3D array to store probabilities at each step 'h' for each cell [i][j]
        double[][][] probabilityMatrix = new double[k + 1][n][n];

        // Initialize the starting position probabilities as 1 (100%)
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                probabilityMatrix[0][i][j] = 1;
            }
        }

        // Array to represent the 8 possible moves of a knight in chess
        int[] directionOffsets = {-2, -1, 2, 1, -2, 1, 2, -1, -2};

        // Iterate over steps from 1 to k
        for (int step = 1; step <= k; ++step) {
            // Iterate over all cells of the chess board
            for (int i = 0; i < n; ++i) {
                for (int j = 0; j < n; ++j) {
                    // Try all 8 possible knight moves
                    for (int move = 0; move < 8; ++move) {
                        // Calculate the new position after the move
                        int newX = i + directionOffsets[move];
                        int newY = j + directionOffsets[move + 1];

                        // Check if the new position is valid (inside the board)
                        if (newX >= 0 && newX < n && newY >= 0 && newY < n) {
                            // Update the probability of the current position after 'step' moves
                            // by adding the probability of the previous position (after 'step - 1' moves)
                            // divided by 8, because a knight has 8 possible moves
                            probabilityMatrix[step][i][j] += probabilityMatrix[step - 1][newX][newY] / 8.0;
                        }
                    }
                }
            }
        }

        // Return the probability of the knight being at position (row, column) after 'k' moves
        return probabilityMatrix[k][row][column];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Calculates the probability of a knight to remain on the chessboard after
    // making 'k' moves starting from the position ('row', 'column').
    double knightProbability(int n, int k, int row, int column) {
        double probability[k + 1][n][n];
        memset(probability, 0, sizeof(probability)); // Initialize the array with 0.

        // At move 0 (the beginning), the probability of being on any cell is 1.
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                probability[0][i][j] = 1;
            }
        }

        // Possible movements for a knight (8 movements)
        int moves[8][2] = {
            {-2, -1}, {-1, -2}, {1, -2}, {2, -1},
            {2, 1}, {1, 2}, {-1, 2}, {-2, 1}
        };

        // Update the probability of each square for each move.
        for (int move = 1; move <= k; ++move) {
            for (int i = 0; i < n; ++i) {
                for (int j = 0; j < n; ++j) {
                    for (int p = 0; p < 8; ++p) {
                        int nextRow = i + moves[p][0];
                        int nextCol = j + moves[p][1];
                        // If the new position is within the board
                        if (nextRow >= 0 && nextRow < n && nextCol >= 0 && nextCol < n) {
                            probability[move][i][j] += probability[move - 1][nextRow][nextCol] / 8.0;
                        }
                    }
                }
            }
        }

        // Return the final probability of the knight staying on the board
        // after 'k' moves from the initial position ('row', 'column').
        return probability[k][row][column];
    }
};

```

Typescript Solution

```typescript
function knightProbability(n: number, k: number, startRow: number, startColumn: number): number {
    // Initialize a 3D array to hold the probabilities of the knight being on a square after h moves.
    const probabilities = new Array(k + 1).fill(0)
        .map(() => new Array(n).fill(0).map(() => new Array(n).fill(0)));

    // Set initial probabilities to 1 for all positions when no move is made (h = 0).
    for (let row = 0; row < n; row++) {
        for (let col = 0; col < n; col++) {
            probabilities[0][row][col] = 1;
        }
    }

    // Define the moves a knight can make (8 possible moves).
    const moves = [-2, -1, 2, 1, -2, 1, 2, -1, -2];

    // Calculate the probabilities after every move from 1 to k.
    for (let move = 1; move <= k; move++) {
        for (let row = 0; row < n; row++) {
            for (let col = 0; col < n; col++) {
                // Add the probabilities of each possible previous position.
                for (let p = 0; p < 8; p++) {
                    const prevRow = row + moves[p];
                    const prevCol = col + moves[p + 1];
                    if (prevRow >= 0 && prevRow < n && prevCol >= 0 && prevCol < n) {
                        probabilities[move][row][col] += probabilities[move - 1][prevRow][prevCol] / 8;
                    }
                }
            }
        }
    }

    // Return the probability that the knight remains on the board after k moves from the starting position.
    return probabilities[k][startRow][startColumn];
}

```

Time and Space Complexity

The time complexity of the code is `O(k * n^2)`. This is because there are `k + 1` layers, each having an `n x n` grid representing the chessboard. Each cell in the grid is visited once for each of the `k` steps. For each cell, we consider all 8 possible moves of a knight, but this constant factor does not affect the overall time complexity.

The space complexity of the code is `O(k * n^2)` as well. This is due to the three-dimensional list `f` that is used to store the probabilities. The size of this list is determined by the number of steps `k + 1` and the size of the chessboard `n x n`.