1197. Minimum Knight Moves

Medium Breadth-First Search Leetcode Link

Problem Description

In this problem, we are given an infinite chess board that can be imagined as an endless grid with coordinates ranging from negative infinity to positive infinity. We have a chess piece—a knight—placed at the origin `[0, 0]` of this grid. The objective is to calculate the minimum number of moves that the knight must make to reach a specific square `[x, y]` on the chessboard.

A knight in chess moves in an L-shape: it can move two squares in one direction (either horizontally or vertically) and then make a 90-degree turn to move one square in a perpendicular direction. This gives the knight a total of eight possible moves at any given point.

The problem requires us to determine the least number of moves necessary to get the knight from its starting position `[0, 0]` to any target coordinates `[x, y]`. The question assures that it is always possible to reach the target square.

Intuition

The solution to this problem is guided by an approach known as Breadth-First Search (BFS), which is a common algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node in a graph, sometimes referred to as a 'search key'), and explores all of the neighbor nodes at the present depth prior to moving on to nodes at the next depth level.

Here's how we apply BFS to solve this problem:

- Start by enqueuing the initial position of the knight (0, 0).
- Explore all possible moves a knight can make from its current position.
- For each move, check if the new position matches the target position `[x, y]`. If so, we return the current number of moves taken to get there as our answer.
- If the new position is not the target and has not been visited yet, add it to a queue for further exploration and mark it as visited.
- Increase the move counter each time we've explored all positions at the current level of depth.
- Repeat these steps until the target position is reached.

This process is efficient for finding the shortest path in an unweighted graph—or in this case, an infinite grid—where the distance between all adjacent nodes is equal.

Solution Approach

The solution uses Breadth-First Search (BFS), which is an algorithm well-suited for searching for the shortest path in an unweighted graph. In this case, the graph can be thought of as an infinite 2D grid where each cell is a node and each knight's move represents an edge connecting two nodes.

Breadth-First Search (BFS)

The BFS algorithm works level by level. Starting from a source node, BFS examines all neighbor nodes at the current depth before moving on to nodes at the next depth level. This method ensures that the path found to any node is the shortest one.

Data Structures Used

To implement BFS:

1. **Queue `q`**: A double-ended queue (`deque` in Python) is used to store nodes to explore in the order they were encountered.
2. **Set `vis`**: A set is used to keep track of visited nodes to prevent re-processing them.

Algorithm Steps

1. **Initialization**: Add the starting position, `[0, 0]`, to the `q` queue and mark it as visited by adding it to the `vis` set. Initialize a step counter `ans` to zero.
2. **Processing Nodes**: Continue the process while there are nodes in the queue to explore.
   - For each iteration (or level in the BFS), check all nodes currently in the queue.
   - Remove a node from the front of the queue using `popleft()`.
   - If this node is the target `[x, y]`, return the `ans` as the minimum number of steps.
3. **Exploring Neighbors**: For the current node at position `(i, j)`, calculate all possible positions where the knight can move based on the defined moves in `dirs`.
   - If a neighbor node `(c, d)` has not been visited, mark it as visited by adding it to the `vis` set and append it to the queue `q`.
4. **Incrementing Steps**: After exploring all nodes at the current depth, increment the `ans` counter by 1 before moving on to nodes at the next depth level.
5. **Termination**: If the target node `[x, y]` is reached, the current value of `ans` will be the minimum number of steps needed, and the function returns this value.

The loop continues until we reach the target node, at which point the function exits with the answer. The BFS guarantees that when we reach `[x, y]`, it will be the shortest path due to the way BFS explores all paths of `n` length before moving to paths of `n+1` length.

By using a set `vis`, the algorithm ensures each node is processed only once, avoiding redundant calculations and cycles which is crucial for efficiency on an infinite grid.

Overall, the BFS approach is efficient and guarantees that the shortest path will be found in a scenario like this where each move is considered of equal 'weight' or distance.

Example Walkthrough

Let's go through a small example to illustrate how the solution approach works. Suppose we want to find the minimum moves for a knight to reach `[2, 1]` from `[0, 0]`.

1. **Initialization**: We start by initializing our BFS. We enqueue the starting position `[0, 0]` into our queue `q` and add it to our visited set `vis`. The step counter `ans` is also initialized to `0`.
2. **Processing Nodes**: At the first level of BFS (starting with `ans = 0`), the queue `q` contains just `[0, 0]`.
3. **Exploring Neighbors**: From `[0, 0]`, a knight can move to eight possible positions: `[2, 1]`, `[1, 2]`, `[-1, 2]`, `[-2, 1]`, `[-2, -1]`, `[-1, -2]`, `[1, -2]`, and `[2, -1]`. We add each of these to the queue `q` (if not already visited) and add them to the visited set `vis`.
4. **Incrementing Steps**: There is no need to increment `ans` yet because we may find our target coordinates at this level. However, none of the positions we have right now from `[0, 0]` are the target.
5. **Identifying the Target**: The target `[2, 1]` is indeed one of the potential moves directly from `[0, 0]`. As soon as we discover this during our search of neighbors, we know that the minimum moves to reach `[2, 1]` from `[0, 0]` is one.
6. **Termination**: Since we have reached our target `[2, 1]`, we can return `ans` which still holds the value `0`. However, as we move directly from the starting point `[0, 0]` to the target `[2, 1]`, we conclude that the number of moves required is `1`.

This is a simple and ideal case where we found the target in the first set of moves from the starting position. In cases where the target position is not one of the initial possible moves, we continue the BFS iteration, each time exploring all possible moves from the positions currently in the queue, incrementing `ans` after all possible moves of the current `ans` have been explored, and repeating the process until the target `[x, y]` is found. The BFS ensures we find the shortest path due to its level-by-level exploration.

Python Solution

```python
from collections import deque

class Solution:
    def minKnightMoves(self, x: int, y: int) -> int:
        # Initialize a queue and start with the knight's initial position (0, 0).
        queue = deque([(0, 0)])

        # The initial number of moves is set to zero.
        moves_count = 0

        # Use a set to keep track of the visited positions to prevent revisits.
        visited = {(0, 0)}

        # The possible moves a knight can take: 8 directions.
        directions = ((-2, 1), (-1, 2), (1, 2), (2, 1), (2, -1), (1, -2), (-1, -2), (-2, -1))

        # Run BFS until the queue is empty.
        while queue:
            # Process nodes level by level.
            for _ in range(len(queue)):
                current_i, current_j = queue.popleft()  # Current position

                # If the target position is reached, return the number of moves.
                if (current_i, current_j) == (x, y):
                    return moves_count

                # Explore all possible moves from the current position
                for delta_i, delta_j in directions:
                    new_i, new_j = current_i + delta_i, current_j + delta_j

                    # If the new position is not yet visited, mark it as visited and add to queue.
                    if (new_i, new_j) not in visited:
                        visited.add((new_i, new_j))
                        queue.append((new_i, new_j))

            # Increment the number of moves after expanding all nodes at the current level.
            moves_count += 1

        # If we exit the while loop something went wrong, we should never reach here.
        return -1

```

Java Solution

```java
import java.util.ArrayDeque;
import java.util.Queue;

class Solution {
    public int minKnightMoves(int x, int y) {
        // Offset coordinates by 310 to deal with negative indices since a knight can move backward.
        x += 310;
        y += 310;

        // Starting moves count from (0, 0) which is at the center after offset
        int moves = 0;

        // Initialize queue for BFS and add starting position after offset
        Queue<int[]> queue = new ArrayDeque<>();
        queue.offer(new int[] {310, 310});

        // Visited matrix to keep track of already visited points.
        boolean[][] visited = new boolean[621][621];
        visited[310][310] = true;

        // Directions a knight can move: 8 possibilities
        int[][] directions = {{-2, 1}, {-1, 2}, {1, 2}, {2, 1}, {2, -1}, {1, -2}, {-1, -2}, {-2, -1}};

        while (!queue.isEmpty()) {
            // Number of elements in the current level
            for (int i = queue.size(); i > 0; --i) {
                // Poll the first element in the queue
                int[] point = queue.poll();

                // Check if the current position is the target position
                if (point[0] == x && point[1] == y) {
                    return moves;
                }

                // Explore all possible moves from the current position
                for (int[] direction : directions) {
                    int nextX = point[0] + direction[0];
                    int nextY = point[1] + direction[1];

                    // Make sure the new position is within bounds and hasn't been visited
                    if (nextX >= 0 && nextY >= 0 && nextX < visited.length && nextY < visited[nextX].length && !visited[nextX][next
                        visited[nextX][nextY] = true;
                        queue.offer(new int[] {nextX, nextY});
                    }
                }
            }
            // Increment moves after finishing all moves of the current level
            ++moves;
        }
        // Return -1 if we never reach the destination position (should not happen with correct logic)
        return -1;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <queue>
#include <utility>

class Solution {
public:
    int minKnightMoves(int x, int y) {
        // Offset the x and y to avoid negative index issues.
        // The number 310 is chosen to handle negative coordinates
        // because a knight cannot be more than 310 moves away from the origin in any direction.
        x += 310;
        y += 310;

        // Initialize the answer to 0, which represents the number of moves.
        int minMoves = 0;
        std::queue<std::pair<int, int>> queue; // Queue to manage BFS.
        queue.push({310, 310}); // Starting point (0,0) with the offset.

        // Visited matrix to keep track of visited cells.
        std::vector<std::vector<bool>> visited(700, std::vector<bool>(700, false));
        visited[310][310] = true; // Mark the starting point as visited.

        // Directions a knight can move on a chessboard.
        std::vector<std::vector<int>> directions = {
            {-2, 1}, {-1, 2}, {1, 2}, {2, 1},
            {2, -1}, {1, -2}, {-1, -2}, {-2, -1}
        };

        // BFS Algorithm.
        while (!queue.empty()) {
            // Traverse nodes level by level.
            for (int size = queue.size(); size > 0; --size) {
                auto current = queue.front();
                queue.pop();

                // Check if we have reached the target (x, y) cell.
                if (current.first == x && current.second == y) return minMoves;

                // Explore all possible moves from the current position.
                for (auto& direction : directions) {
                    int nextRow = current.first + direction[0];
                    int nextCol = current.second + direction[1];

                    // If the cell is not visited, mark it visited and add to the queue.
                    if (!visited[nextRow][nextCol]) {
                        visited[nextRow][nextCol] = true;
                        queue.push({nextRow, nextCol});
                    }
                }
            }
            // Increment the number of moves after exploring all positions in current level.
            ++minMoves;
        }

        // If the function hasn't returned yet, something went wrong.
        // With the BFS approach, however, we should always be able to return before hitting this line.
        return -1;
    }
};

```

Typescript Solution

```typescript
type Point = [number, number];

// Offset to handle negative coordinates. A knight cannot be more than 310 moves away from the origin.
const OFFSET = 310;
const BOARD_SIZE = 700; // Define board as 700x700 to include the offset space.

let visited: boolean[][] = Array.from({ length: BOARD_SIZE }, () => Array(BOARD_SIZE).fill(false));

const directions: Point[] = [
  [-2, 1], [-1, 2], [1, 2], [2, 1],
  [2, -1], [1, -2], [-1, -2], [-2, -1],
];

function minKnightMoves(x: number, y: number): number {
    // Normalize the target coordinates with the offset.
    x += OFFSET;
    y += OFFSET;

    let minMoves: number = 0;
    let queue: Point[] = [[OFFSET, OFFSET]]; // Initialize BFS queue with the starting point (0, 0) with the offset.
    visited[OFFSET][OFFSET] = true; // Mark the starting point as visited.

    while (queue.length > 0) {
        // Number of nodes at the current BFS level.
        let levelSize: number = queue.length;

        // Traverse nodes level by level.
        for (let i = 0; i < levelSize; i++) {
            let current: Point = queue.shift()!; // Retrieve and remove the first element.

            // Check if the current position is the target position.
            if (current[0] === x && current[1] === y) {
                return minMoves;
            }

            // Explore all possible moves from current position.
            directions.forEach(direction => {
                let nextRow: number = current[0] + direction[0];
                let nextCol: number = current[1] + direction[1];

                // Check if the cell is within the board limits and not visited.
                if (nextRow >= 0 && nextRow < BOARD_SIZE && nextCol >= 0 && nextCol < BOARD_SIZE && !visited[nextRow][nextCol]) {
                    visited[nextRow][nextCol] = true;
                    queue.push([nextRow, nextCol]);
                }
            });
        }
        // Increment the number of moves after exploring the current level.
        minMoves++;
    }

    // Return -1 if no solution found, though BFS should always find the solution.
    return -1;
}

// Reset visited matrix for a fresh computation.
function resetVisited(): void {
    visited = visited.map(row => row.fill(false));
}

```

Time and Space Complexity

The given Python code implements a breadth-first search (BFS) algorithm to find the minimum number of moves a knight can take to reach a given position `(x, y)` on an infinite chessboard, starting from position `(0, 0)`. The time complexity and space complexity of this BFS algorithm are as follows:

Time Complexity

To analyze the time complexity, let's consider the increments to coordinates as potential moves from one square to another. For each move, we have 8 possible directions in which the knight can move. The BFS algorithm ensures that every position is visited only once, thanks to the `vis` set which tracks the visited positions.

Since the board is infinite, the maximum distance from the origin in terms of the number of moves can be represented by `max(abs(x), abs(y))`. This is because, in the worst-case scenario, we can consider moving diagonally (in L-shaped movements) towards the target, which is roughly `max(abs(x), abs(y))` moves away. However, the actual number of moves requires considering the peculiarities of knight's movements. The BFS will have a branching factor of at most 8 (the possible moves the knight can make), and the depth will be proportional to the distance from the origin to the target.

Thus, the time complexity can be approximated as `O(8^(d))` where `d` is the depth of the BFS, or more accurately, `O((max(abs(x), abs(y)))^2)` because each layer of BFS (which corresponds to one knight's move) potentially adds up to 8 new positions in the queue.

Space Complexity

The space complexity is primarily dictated by the storage required for the `vis` set and the `q` queue. The `vis` set contains all the unique positions we have visited during the BFS.

- The space taken by the `vis` set is proportional to the number of elements in it, which is the number of unique positions the algorithm will visit, roughly the same as the time complexity, leading to an `O((max(abs(x), abs(y)))^2)` space complexity.
- The `q` queue stores the positions that need to be explored. In the worst case, the queue could store all of the positions that we visit. Hence, the space complexity contributed by the `q` is similar to the vis set, `O((max(abs(x), abs(y)))^2)`.

Overall, the space complexity of the algorithm is `O((max(abs(x), abs(y)))^2)`.