1197. Minimum Knight Moves Medium Breadth-First Search

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

Leetcode Link

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

2. Set vis: A set is used to keep track of visited nodes to prevent re-processing them.

2. Processing Nodes: Continue the process while there are nodes in the queue to explore.

If this node is the target [x, y], return the ans as the minimum number of steps.

Remove a node from the front of the queue using popleft().

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.

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.

Data Structures Used

an edge connecting two nodes.

Breadth-First Search (BFS)

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

For each iteration (or level in the BFS), check all nodes currently in the queue.

the next depth level.

function returns this value.

crucial for efficiency on an infinite grid.

considered of equal 'weight' or distance.

set vis. The step counter ans is also initialized to 0.

none of the positions we have right now from [0, 0] are the target.

Example Walkthrough

counter ans to zero.

on the defined moves in dirs. o 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.

3. Exploring Neighbors: For the current node at position (i, j), calculate all possible positions where the knight can move based

4. Incrementing Steps: After exploring all nodes at the current depth, increment the ans counter by 1 before moving on to nodes at

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

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

Overall, the BFS approach is efficient and guarantees that the shortest path will be found in a scenario like this where each move is

The loop continues until we reach the target node, at which point the function exits with the answer. The BFS guarantees that when

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

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,

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

2. Processing Nodes: At the first level of BFS (starting with ans = 0), the queue q contains just [0, 0].

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.

Python Solution

class Solution:

10

13

14

15

16

17

18

19

30

31

32

33

34

35

36

37

38

39

40

41

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

29

30

31

32

33

34

35

36

37

38

39

40

41

42

49

50

51

52

53

54

55

56

57

58

59

60

12];

13

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

58

59

60

};

++minMoves;

return -1;

Typescript Solution

const OFFSET = 310;

x += OFFSET;

y += OFFSET;

1 type Point = [number, number];

const directions: Point[] = [

[-2, 1], [-1, 2], [1, 2], [2, 1],

let minMoves: number = 0;

while (queue.length > 0) {

[2, -1], [1, -2], [-1, -2], [-2, -1],

function minKnightMoves(x: number, y: number): number {

// Normalize the target coordinates with the offset.

// Number of nodes at the current BFS level.

if (current[0] === x && current[1] === y) {

let levelSize: number = queue.length;

for (let i = 0; i < levelSize; i++) {

// Traverse nodes level by level.

// Reset visited matrix for a fresh computation.

visited = visited.map(row => row.fill(false));

once, thanks to the vis set which tracks the visited positions.

from collections import deque

visited = $\{(0, 0)\}$

moves_count += 1

public int minKnightMoves(int x, int y) {

Queue<int[]> queue = new ArrayDeque<>();

boolean[][] visited = new boolean[621][621];

int[] point = queue.poll();

return moves;

// Directions a knight can move: 8 possibilities

// Poll the first element in the queue

if (point[0] == x && point[1] == y) {

for (int[] direction : directions) {

int nextX = point[0] + direction[0];

int nextY = point[1] + direction[1];

visited[nextX][nextY] = true;

queue.offer(new int[] {310, 310});

visited[310][310] = true;

return -1

1 import java.util.ArrayDeque;

x += 310;

y += 310;

int moves = 0;

2 import java.util.Queue;

4 class Solution {

Java Solution

while queue:

def minKnightMoves(self, x: int, y: int) -> int:

Run BFS until the queue is empty.

Process nodes level by level.

Use a set to keep track of the visited positions to prevent revisits.

directions = ((-2, 1), (-1, 2), (1, 2), (2, 1), (2, -1), (1, -2), (-1, -2), (-2, -1))

The possible moves a knight can take: 8 directions.

if (new_i, new_j) not in visited:

visited.add((new_i, new_j))

queue.append((new_i, new_j))

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.

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

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

20 for _ in range(len(queue)): 21 current_i, current_j = queue.popleft() # Current position 22 23 # If the target position is reached, return the number of moves. 24 if (current_i, current_j) == (x, y): 25 return moves_count 26 27 # Explore all possible moves from the current position 28 for delta_i, delta_j in directions: 29 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.

Increment the number of moves after expanding all nodes at the current level.

// Offset coordinates by 310 to deal with negative indices since a knight can move backward.

int[][] directions = $\{\{-2, 1\}, \{-1, 2\}, \{1, 2\}, \{2, 1\}, \{2, -1\}, \{1, -2\}, \{-1, -2\}, \{-2, -1\}\};$

// 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</pre>

If we exit the while loop something went wrong, we should never reach here.

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

// Check if the current position is the target position

// Explore all possible moves from the current position

// Initialize queue for BFS and add starting position after offset

// Visited matrix to keep track of already visited points.

```
23
24
           while (!queue.isEmpty()) {
25
               // Number of elements in the current level
26
               for (int i = queue.size(); i > 0; --i) {
27
28
```

```
43
                             queue.offer(new int[] {nextX, nextY});
 44
 45
 46
                 // Increment moves after finishing all moves of the current level
 47
 48
                 ++moves;
 49
             // Return -1 if we never reach the destination position (should not happen with correct logic)
 50
 51
             return -1;
 52
 53
 54
C++ Solution
   #include <vector>
  2 #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
  9
             // because a knight cannot be more than 310 moves away from the origin in any direction.
 10
             x += 310;
 11
             y += 310;
 12
 13
             // Initialize the answer to 0, which represents the number of moves.
 14
 15
             int minMoves = 0;
             std::queue<std::pair<int, int>> queue; // Queue to manage BFS.
 16
             queue.push({310, 310}); // Starting point (0,0) with the offset.
 17
 18
 19
             // Visited matrix to keep track of visited cells.
 20
             std::vector<std::vector<bool>> visited(700, std::vector<bool>(700, false));
 21
             visited[310][310] = true; // Mark the starting point as visited.
 22
             // Directions a knight can move on a chessboard.
 23
             std::vector<std::vector<int>> directions = {
 24
 25
                 \{-2, 1\}, \{-1, 2\}, \{1, 2\}, \{2, 1\},
                 \{2, -1\}, \{1, -2\}, \{-1, -2\}, \{-2, -1\}
 26
 27
             };
 28
 29
             // BFS Algorithm.
 30
             while (!queue.empty()) {
 31
                 // Traverse nodes level by level.
 32
                 for (int size = queue.size(); size > 0; --size) {
 33
                     auto current = queue.front();
 34
                     queue.pop();
 35
 36
                     // Check if we have reached the target (x, y) cell.
 37
                     if (current.first == x && current.second == y) return minMoves;
 38
                     // Explore all possible moves from the current position.
 39
                     for (auto& direction: directions) {
 40
 41
                         int nextRow = current.first + direction[0];
 42
                         int nextCol = current.second + direction[1];
 43
 44
                         // If the cell is not visited, mark it visited and add to the queue.
 45
                         if (!visited[nextRow][nextCol]) {
 46
                             visited[nextRow][nextCol] = true;
 47
                             queue.push({nextRow, nextCol});
 48
```

// Increment the number of moves after exploring all positions in current level.

// Offset to handle negative coordinates. A knight cannot be more than 310 moves away from the origin.

let current: Point = queue.shift()!; // Retrieve and remove the first element.

// Check if the cell is within the board limits and not visited.

let queue: Point[] = [[OFFSET, OFFSET]]; // Initialize BFS queue with the starting point (0, 0) with the offset.

if (nextRow >= 0 && nextRow < BOARD_SIZE && nextCol >= 0 && nextCol < BOARD_SIZE && !visited[nextRow][nextCol]) {

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

// With the BFS approach, however, we should always be able to return before hitting this line.

// If the function hasn't returned yet, something went wrong.

const BOARD_SIZE = 700; // Define board as 700x700 to include the offset space.

visited[OFFSET] [OFFSET] = true; // Mark the starting point as visited.

// Check if the current position is the target position.

let nextRow: number = current[0] + direction[0];

let nextCol: number = current[1] + direction[1];

// Increment the number of moves after exploring the current level.

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

visited[nextRow][nextCol] = true;

queue.push([nextRow, nextCol]);

33 return minMoves; 34 35 36 // Explore all possible moves from current position. 37 directions.forEach(direction => {

});

minMoves++;

function resetVisited(): void {

Time and Space Complexity

return -1;

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

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

Since the board is infinite, the maximum distance from the origin in terms of the number of moves can be represented by

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.

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

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

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, $0((\max(abs(x), abs(y)))^2)$. Overall, the space complexity of the algorithm is $O((\max(abs(x), abs(y)))^2)$.