Knight Moves (BFS in 2d Grid)

The **Knight Moves** problem requires finding the shortest path a chess knight can take between two squares on an 8×8 chessboard. A knight moves in an L-shape: two squares in one direction and one square perpendicular, or one square in one direction and two squares perpendicular, giving it up to eight possible moves from any position, limited by the board edges. The input consists of multiple test cases, each specifying two squares in algebraic notation (a-h for columns and 1-8 for rows). The output must be formatted as: "To get from [start] to [end] takes [n] knight moves," where [n] is the minimum number of moves. The problem can be modeled as an unweighted graph where each square is a node and knight moves are edges. Using **breadth-first search (BFS)** guarantees finding the shortest path, as BFS explores all positions level by level. Key steps include converting algebraic notation to coordinates, validating moves to stay within the board, tracking visited squares to prevent revisits, and managing a BFS queue to count moves efficiently.

Steps to Solve the Knight Moves Problem

Step 1: Understand the Problem
We need to find the **minimum number of knight moves** to go from one square to another on an 8×8 chessboard. A knight moves in an L-shape: **two squares in one direction and one square perpendicular**, resulting in **eight possible moves** from any position.

Step 2: Set Up Chessboard Representation
Represent the chessboard as an **8×8 grid**. Convert algebraic notation (like "e2") to coordinates:

- Columns a-h → 0-7
- Rows 1-8 → 0-7

Step 3: Define Knight Moves
Use two arrays to represent all possible knight moves:

dx[] = {2, 2, 1, 1, -1, -1, -2, -2};

dy[] = {1, -1, 2, -2, 2, -2, 1, -1};

Step 4: Initialize BFS Components

- Create a **visited matrix** to track explored squares.
- Create a **distance/move matrix** to store move counts.

- Use a **queue** for BFS traversal.
- Store each position as a structure or class containing (x, y) and move count.

Step 5: Handle Special Case
If the start and end positions are the same, immediately return **0 moves**.

Step 6: Implement BFS Algorithm

1. Enqueue the starting position with move count 0.
2. Mark the starting position as visited.
3. While the queue is not empty:
    - Dequeue the current position.
    - For each of the eight possible knight moves:
        - Calculate the new position (nx, ny).
        - Check if the new position is **within bounds** and **not visited**.
        - If the new position equals the destination, **return current moves + 1**.
        - Otherwise, mark as visited, set the distance, and enqueue (nx, ny) with moves + 1.

Step 7: Process Input and Output

- Read input pairs until end of file.
- For each pair, call the BFS function.
- Print the result in the required format:

To get from [start] to [end] takes [n] knight moves.

Step 8: Boundary Validation
Ensure all moves remain within the board:

$0 \leq x < 8$ and $0 \leq y < 8$

Input:

e2 e4

a1 b2

b2 c3

a1 h8

a1 h7

h8 a1

b1 c3

f6 f6

Execution Steps for Each Input

Input: e2 e4

- Start: e2 → (4,1), End: e4 → (4,3)
- BFS initializes queue: [(4,1,0)], visited (4,1)=true
- Generate valid knight moves: (6,2), (6,0), (5,3), (3,3), (2,2), (2,0)
- Next level: (6,2,1) → destination (4,3) found → moves = 2
  Output: To get from e2 to e4 takes 2 knight moves.

Input: a1 b2

- a1 → (0,0), b2 → (1,1)
- BFS explores paths via intermediate squares
- Path: (0,0)→(2,1)→(1,3)→(3,2)→(1,1) → 4 moves
  Output: To get from a1 to b2 takes 4 knight moves.

Input: b2 c3

- b2 → (1,1), c3 → (2,2)

- BFS finds path through one intermediate square
- Path: (1,1)→(2,3)→(2,2) → 2 moves
  Output: To get from b2 to c3 takes 2 knight moves.

Input: a1 h8

- a1 → (0,0), h8 → (7,7)
- BFS explores multiple paths
- Optimal path requires 6 moves
  Output: To get from a1 to h8 takes 6 knight moves.

Input: a1 h7

- a1 → (0,0), h7 → (7,6)
- BFS finds optimal path in 5 moves
  Output: To get from a1 to h7 takes 5 knight moves.

Input: h8 a1

- h8 → (7,7), a1 → (0,0)
- Reverse of a1→h8 → 6 moves
  Output: To get from h8 to a1 takes 6 knight moves.

Input: b1 c3

- b1 → (1,0), c3 → (2,2)
- Direct L-shaped move → 1 move
  Output: To get from b1 to c3 takes 1 knight moves.

Input: f6 f6

- Start = End (5,5) → special case → 0 moves
  Output: To get from f6 to f6 takes 0 knight moves.

Final Output :

To get from e2 to e4 takes 2 knight moves.

To get from a1 to b2 takes 4 knight moves.

To get from b2 to c3 takes 2 knight moves.

To get from a1 to h8 takes 6 knight moves.

To get from a1 to h7 takes 5 knight moves.

To get from h8 to a1 takes 6 knight moves.

To get from b1 to c3 takes 1 knight moves.

To get from f6 to f6 takes 0 knight moves.

Pseudocode:

```
Function knightMoves(start, end):
    # Step 1: Convert algebraic notation to coordinates
    startX, startY = convertToCoordinates(start)
    endX, endY = convertToCoordinates(end)

    # Step 2: Handle special case: start = end
    If startX == endX AND startY == endY:
        Return 0

    # Step 3: Initialize BFS
    visited[8][8] = false
    queue = empty queue
    Enqueue (startX, startY, 0)   # last element is move count
```

```
    visited[startX][startY] = true

    # Step 4: Define knight moves
    dx = [2, 2, 1, 1, -1, -1, -2, -2]
    dy = [1, -1, 2, -2, 2, -2, 1, -1]

    # Step 5: BFS loop
    While queue is not empty:
        x, y, moves = Dequeue queue

        For i = 0 to 7:
            nx = x + dx[i]
            ny = y + dy[i]

            # Step 6: Check board boundaries and visited
            If nx >= 0 AND nx < 8 AND ny >= 0 AND ny < 8 AND NOT visited[nx][ny]:
                If nx == endX AND ny == endY:
                    Return moves + 1

                visited[nx][ny] = true
                Enqueue (nx, ny, moves + 1) into queue

# Step 7: Helper function to convert chess notation to coordinates
Function convertToCoordinates(square):
    column = square[0]  # 'a' to 'h'
    row = square[1]     # '1' to '8'
```

x = column - 'a'   # 0 to 7

y = row - '1'   # 0 to 7

Return (x, y)

 # Step 8: Main program

While there is input:

Read start, end

moves = knightMoves(start, end)

Print "To get from [start] to [end] takes [moves] knight moves."

Here is the solution code for Knight Moves:

https://github.com/Hazra32/Algorithm-Problem/blob/main/BFS/Knight%20Moves/knightmoves.cpp