Task 6: Six knights

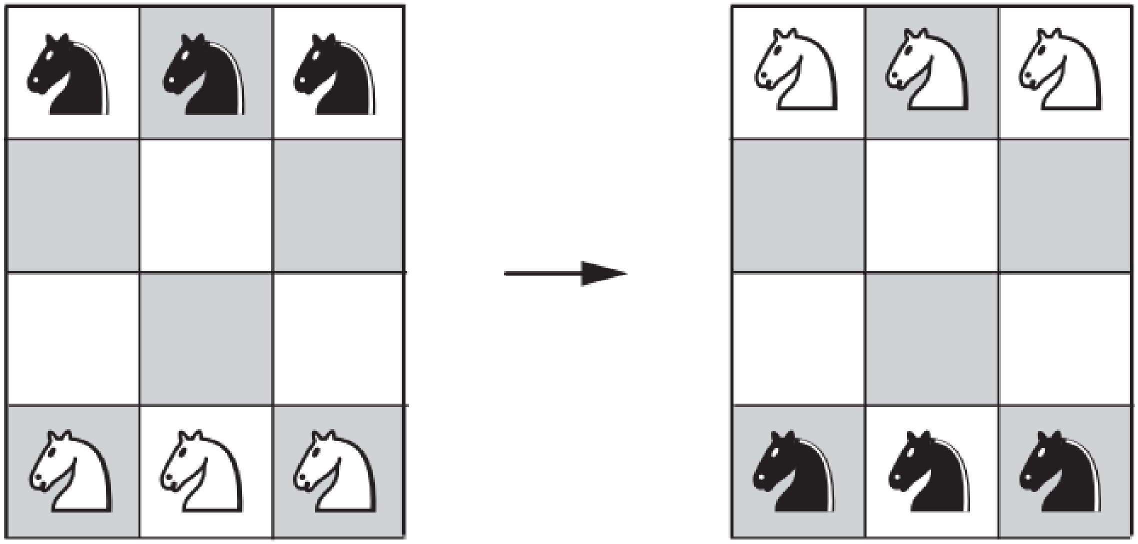
Name	ID
Youssef Wael Hamdy Ibrahim	2001430
Ahmad Youssef Mansour Mahfouz	2002238

Table of Contents

1- Problem description:

There are six knights on a 3 × 4 chessboard: the three white knights are at the bottom row, and the three black knights are at the top row.

Design an iterative improvement algorithm to exchange the knights to get the position shown on the right of the figure in the minimum number of knights moves, not allowing more than one knight on a square at any time.



2- Detailed assumptions:

1) A chessboard of 4X3 is assumed.
2) Knights must follow an L-shaped pattern.
3) A knight can't go outside the board boundaries.
4) Each Knight placement in a board is a State.
5) Best first search is implemented using heuristics.
6) The heuristic is implemented using Manhattan distance.
7) Once in a path you can't visit the same state twice.

3- Detailed solution including the pseudo-code and the description of your solution:

Detailed solution:

1) An initial state is passed in the code.
2) The code starts by Generating moves for knights starting with the white knights.
3) Each state is stored inside a path and each path is stored inside priority queue that is sorted based on heuristic value.
4) The heuristic value of the path is the heuristic value of the last state of this path.
5) The top of the queue (the one with the lowest heuristic value) is hypothetically the closest board arrangement to our solution.
6) If our board arrangement is found its Heuristic will equal -1

Pseudo-code: Link

```
FUNCTION calculateHeuristic(board)
// Inputs: Board is the 2D vector representing the game board
// Output: Returns the heuristic value calculated based on the board state
    heuristic <= 0

    // Define goal positions for white and black knights
    whiteGoalPositions <= {{0, 0}, {0, 1}, {0, 2}}
    blackGoalPositions <= {{3, 0}, {3, 1}, {3, 2}}

    FOR i FROM 0 TO ROWS - 1 DO
        FOR j FROM 0 TO COLS - 1 DO
            IF board[i][j] == WHITE THEN
                minDist <= INT_MAX
                FOR EACH goalPos IN whiteGoalPositions DO
                    dist <= ABS(i - goalPos.first) + ABS(j - goalPos.second)
                    minDist <= MIN(minDist, dist)
                END FOR
                heuristic += minDist
            ELSE IF board[i][j] == BLACK THEN
                minDist <= INT_MAX
                FOR EACH goalPos IN blackGoalPositions DO
                    dist <= ABS(i - goalPos.first) + ABS(j - goalPos.second)
                    minDist <= MIN(minDist, dist)
                END FOR
                heuristic += minDist
            END IF
        END FOR
    END FOR

    IF isGoalState(board) THEN
        RETURN -1
    END IF

    RETURN heuristic / 3.0 + 3
END FUNCTION


FUNCTION isGoalState(board)
// Inputs: board is the 2D vector representing the game board
// Output: Returns boolean true if the board represents a goal state
    RETURN (board[0][0] == WHITE AND board[0][1] == WHITE AND board[0][2] == WHITE) AND (board[3][0] ==
BLACK AND board[3][1] == BLACK AND board[3][2] == BLACK)
END FUNCTION

FUNCTION getKnightMoves(x, y, board)
// Inputs: x is the X coordinate of the knight, y is the Y coordinate of the knight, board is the 2D
vector representing the game board
// Output: Returns a list of valid knight moves from the given position
    moves <= []

    directions <= {{1, 2}, {-2, 1}, {-2, -1}, {-1, -2}, {1, -2}, {2, -1}, {2, 1}, {-1, 2}}

    FOR EACH dir IN directions DO
        newX <= x + dir.first
        newY <= y + dir.second
        IF isEmptyAndWithinLimits(board, newX, newY) THEN
            moves.push({newX, newY})
        END IF
    END FOR

    RETURN moves
END FUNCTION

FUNCTION generateNextState(currentState, pq, currentPath)
// Inputs: currentState is the current state of the game, pq is the priority queue of paths, currentPath
is the current path
// Output: Generates and pushes next path to the priority queue
    FOR i FROM 0 TO ROWS - 1 DO
        FOR j FROM 0 TO COLS - 1 DO
            IF currentState.board[i][j] != 0 THEN
                moves <= getKnightMoves(i, j, currentState.board)
                FOR EACH move IN moves DO
                    nextBoard <= COPY currentState.board
                    SWAP nextBoard[i][j] AND nextBoard[move.first][move.second]
                    nextHeuristic <= calculateHeuristic(nextBoard)
                    nextState <= State(nextBoard, nextHeuristic)
                    IF NOT currentPath.hasDupeStates(nextState) THEN
                        currentPath.push(nextState)
                        pq.push(currentPath)
                    END IF
                END FOR
            END IF
        END FOR
    END FOR
END FUNCTION

// Performs Best-First Search to find the solution
FUNCTION bestFirstSearch(initialState)
// Inputs: initialState is the initial state of the game
// Output: Prints Path leading to solution if found
    pq <= Priority Queue of Paths
    initialHeuristic <= calculateHeuristic(initialState)
    beginningState <= State(initialState, initialHeuristic)
    initialPath <= Path(beginningState)
    pq.push(initialPath)

    WHILE NOT pq.empty() DO
        currentPath <= pq.top()
        currentState <= currentPath.path[currentPath.path.size() - 1]
        pq.pop()

        IF isGoalState(currentState.board) OR currentState.heuristic == -1 THEN
            PRINT "\n\nSolution found:"
            printBoard(currentState.board)
            PRINT "Path: "
            currentPath.printPath()
            PRINT "\nTotal moves: " + currentPath.length() - 1
            PRINT "\nTotal paths: " + pq.size()
            RETURN
        END IF

        IF currentPath.length() <= MAX_MOVES THEN
            generateNextState(currentState, pq, currentPath)
        END IF
    END WHILE

    PRINT "No solution found."
END FUNCTION

FUNCTION withinLimits(x, y)
// Inputs: x is the X coordinate of the knight, y is the Y coordinate of the knight
// Output: Returns boolean true if the coordinates are within the board limits
    RETURN (x >= 0 AND y >= 0) AND (x < ROWS AND y < COLS)
END FUNCTION

FUNCTION isEmptyAndWithinLimits(board, x, y)
// Inputs: board is the 2D vector representing the game board, x is the X coordinate of the knight, y is
the Y coordinate of the knight
// Output: Returns boolean true if the knight movement is within the board and the target square is empty
    RETURN withinLimits(x, y) AND board[x][y] == 0
END FUNCTION
```

4- complexity analysis for algorithm:

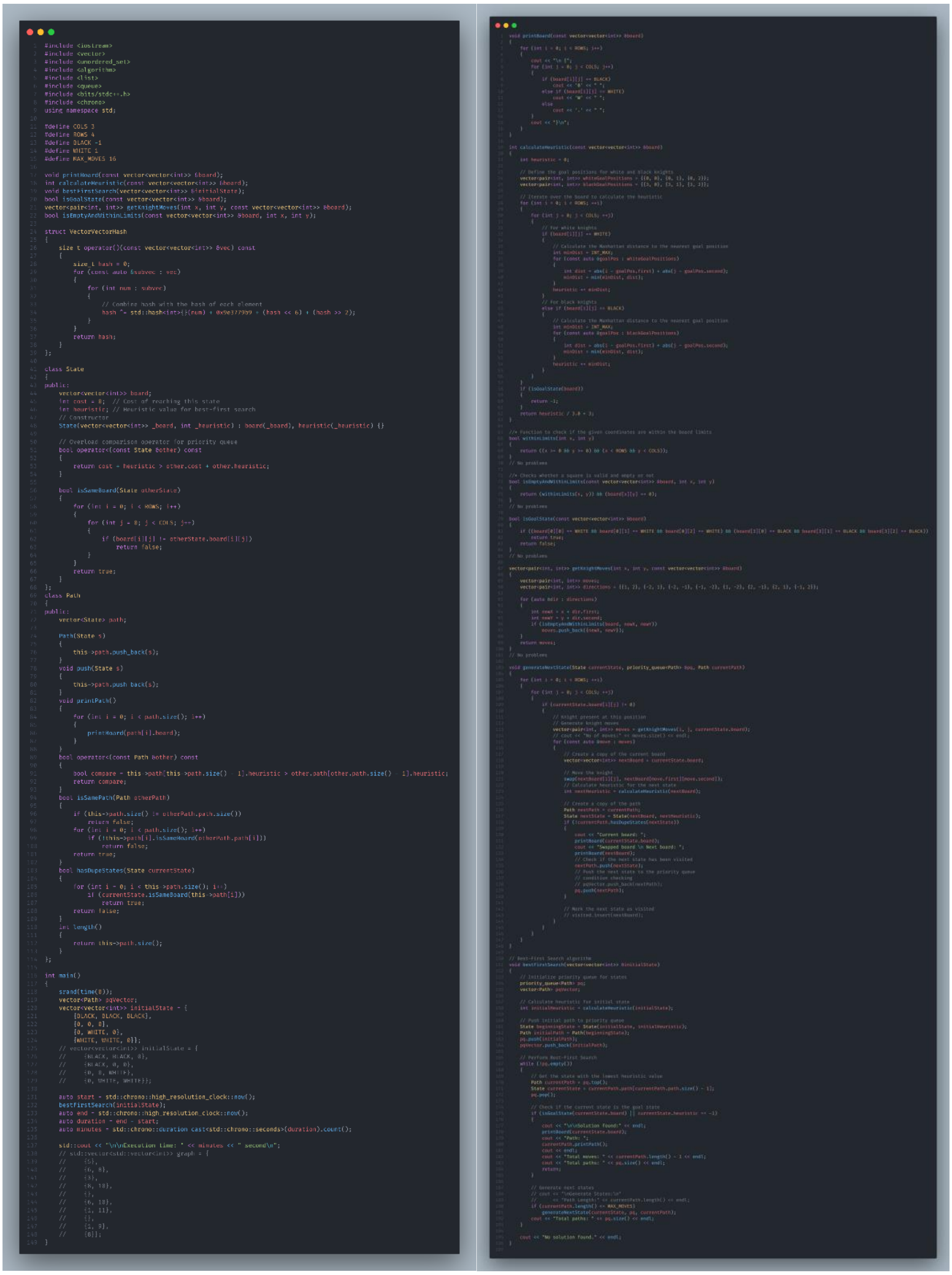
Time Complexity: O(N * Log(N)).

Space complexity: O(N).

Where N is the number of paths in priority queue.

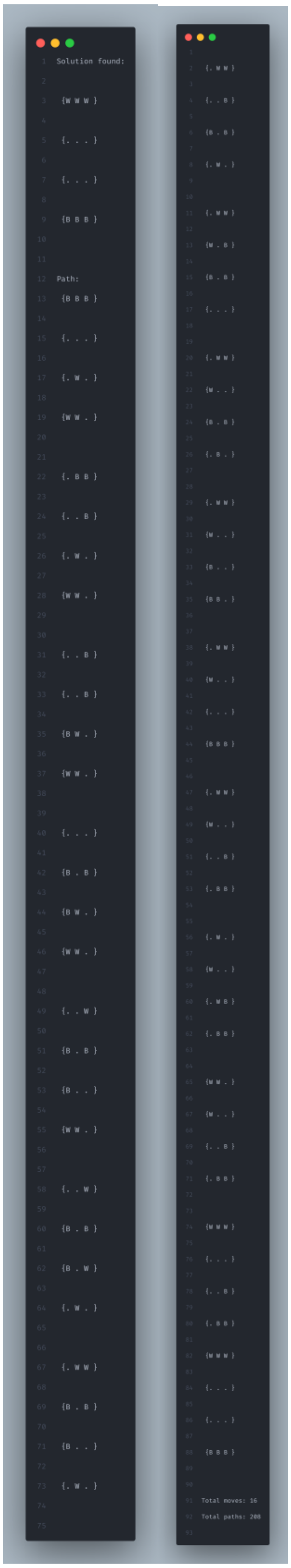
5- Code:

[Link]



MAX_MOVES needs to be updated based on initial state passed.

6- Sample output:



Each state Generated in turn generates more states each chain of states is a path so this code starts with initial state generates a new state and adds it to this state and it's a path so this path is put into Priority queue where it's sorted based on Heuristic values of each state each new iteration it takes top path of queue and then it generates new states for it and so on till we reach goal.

7- comparison with another algorithm:

	Brute force Approach	Best first search
Advantages	Guarantees to find a solution if one exists	A faster approach then brute force and can work with any size of chessboard
Disadvantages	A huge time complexity	Not guaranteed to find a solution if heuristic is poorly chosen
Time Complexity	8^N where N is the number of knights.	O(N*Log(N)) where N is the number of paths in Priority queue

8- conclusion:

Solving the Six knights problem requires one of 2 approaches Brute force, Searching algorithms (best first, Hill climbing, A*).

The brute force method Generates all possible moves combinations till it reaches the goal, but can yield very high time complexity, and isn't feasible for bigger chessboards.

Best first algorithm is better than brute force if heuristic calculations are good and can reach solution a lot faster than brute force approach, this approach is an approximation so it might not get the best path each time and it might encounter convergence problems.

9- References:

- Best first search.
- Wiki.
- Puzzle stack exchange.
- Princeton university pages 3,4 and 5.
- Carnegie Mellon University.