Assignment 3
The Knight's Tour problem as a CSP

The knight in chess is the piece that is shaped like a horse's head. The knight can move in an "L-shaped" pattern, advancing two squares in one direction and then having the option to turn either right or left. If we display all the current possible moves for the knight, the resulting pattern is as follows (see Figure 1).

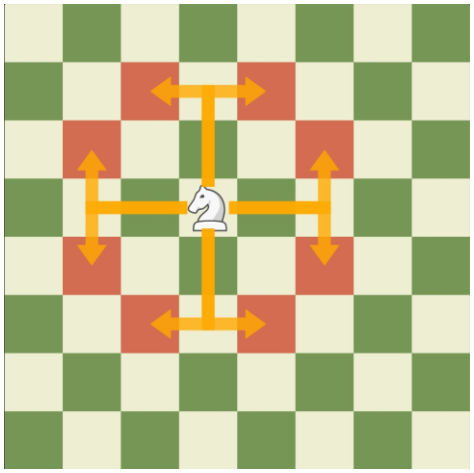


Figure 1 All knight's possible moves

The knight can move to the square marked in red and then repeat the process on the new square. A knight's tour is a sequence of moves that a knight can make to visit every chess square exactly once.

The objective of this assignment is to tackle the knight's tour problem using backtracking algorithm. For this purpose, we will create the following classes:

1. **Knight class:** Each knight will store these variables:

- ✧ **position:** the coordinates (x, y) for the knight's current position.
- ✧ **assignment:** the sequence of moves taken by the knight.
- ✧ **path:** the list of knight's positions after applying the moves defined in the assignment.

In the Knight class, the following functions should be defined:

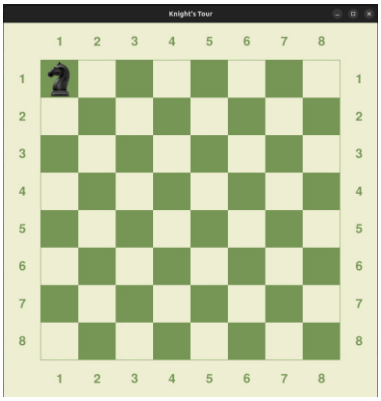
- ✧ **init ():** This function creates a new knight. Initially, the knight's assignment is an empty list. Additionally, this function sets the current position of the knight to (0, 0) and saves the initial position into the path list.

- ✧ **move_forward (direction):** This function moves the knight in one of the 8 directions (***1: up-right, 2: right-up, 3: right-down, 4: down-right, 5: down-left, 6: left-down, 7: left-up, 8: up-left***). It generates the new position of the knight after applying a move.
- ✧ **move_backward (direction):** This function allows the knight to trace back if the applied move is illegal.
- ✧ **consistent (direction):** This function checks the validity of last move added in the assignment. A move is considered invalid if, when applied, it places the knight outside the chessboard or in a position that has already been visited. If a move is illegal, this function will return False.
- ✧ **addMove (direction):** This function adds a move to the assignment, moves the knight using **move_forward (direction)**, and adds the new position to the path.
- ✧ **removeMove (direction):** This function removes the last added move from the assignment, moves the knight backward using **move_backward (direction)**, and removes the last added position from the path.

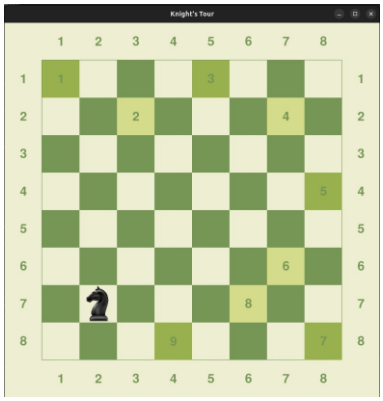
2. Implement the backtracking algorithm following the pseudo-code provided in the course and using the knight class and its methods.

3. **The main function:** The main function applies a run of the backtracking algorithm and displays the optimal solution on an interface, as illustrated in Figure 2.

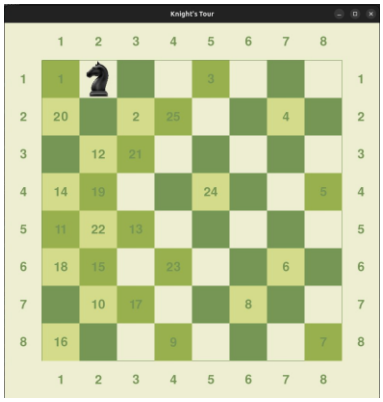
```
def main ( ):

    knight = Knight()
    knight = backtracking(knight)
    print (knight.assignment)
    print (knight.path)

if __name__ == "__main__":
    main()
```



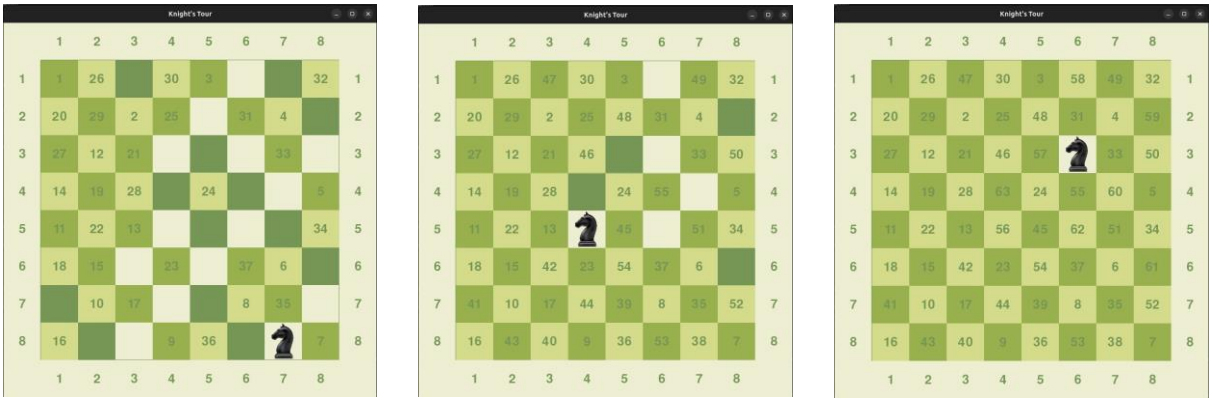
(a)



(b)



(c)



(d) (e) (f)

Figure 2 visualization of a solution of the knight's tour problem using the backtracking algorithm