

# DAYANANDA SAGAR UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING  
SCHOOL OF ENGINEERING  
DAYANANDA SAGAR UNIVERSITY  
KUDLU GATE  
BANGALORE - 560068



## MINI PROJECT REPORT

*ON*

## KNIGHT'S TOUR USING BACKTRACKING ALGORITHM

SUBMITTED TO THE 4<sup>th</sup> SEMESTER ALGORITHM  
DESIGN & ANALYSIS LABORATORY  
BACHELOR OF TECHNOLOGY

*IN*

COMPUTER SCIENCE & ENGINEERING

*Submitted by*

LALITH SAGAR J (ENG18CS0147)

KIRAN B Y (ENG18CS0136)

MAJUNU G (ENG10CS0157)

LALITH ANAND S (ENG18CS0146)

*Under the supervision of*  
**Prof. Shamanth Nagaraj**

# DAYANANDA SAGAR UNIVERSITY

School of Engineering, Kudlu Gate, Bangalore-560068



## CERTIFICATE

*This is to certify that the Algorithm Design & Analysis Mini-Project report entitled "The Knight's Tour using Backtracking algorithm" being submitted to Department of Computer Science and Engineering, School of Engineering, Dayananda Sagar University, Bangalore, for the 4<sup>th</sup> semester B.Tech C.S.E of this university during the academic year 2019-2020.*

Date \_\_\_\_\_

Signature of the faculty in-charge: \_\_\_\_\_

Max Marks	Marks Obtained

\_\_\_\_\_  
Signature of Chairman

Department of Computer Science & Engineering

## DECLARATION

We hereby declare that the work presented in this mini project entitled - “***The Knight’s Tour using Backtracking algorithm*** “, has been carried out by us and it has not been submitted for the award of any degree, diploma or the mini project of any other college or university.

### Team members:

LALITH SAGAR J (ENG18CS0147)  
KIRAN B Y (ENG18CS0136)  
MAJUNU G (ENG18CS0157)  
LALITH ANAND S (ENG18CS0146)

## ACKNOWLEDGEMENT

The satisfaction that accompanies the successful completion of task would be incomplete without the mention of the people who made it possible and whose constant guidance and encouragement crown all the efforts with success.

We are especially thankful to our **Chairman, Dr. M K Banga** for providing necessary departmental facilities, moral support and encouragement.

We are very much thankful to Prof. **Shamanth Nagaraj**, for providing help and suggestions in completion of this mini project successfully.

We have received a great deal of guidance and co-operation from our friends and we wish to thank all that have directly or indirectly helped us in the successful completion of this project work.

## **TABLE OF CONTENTS**

<b><u>Contents</u></b>	<b><u>Page no</u></b>
Abstract	<b>6</b>
Introduction	<b>7</b>
Problem Statement	<b>9</b>
Objective	<b>11</b>
Methodology	<b>12</b>
Algorithm	<b>13</b>
Source Code	<b>14</b>
Time Complexity	<b>16</b>
GUI for problem	<b>17</b>
Results	<b>19</b>
Conclusion	<b>21</b>
Reference	<b>22</b>

## **ABSTRACT**

A knight's tour is a sequence of moves of a knight on a chessboard such that the knight visits every square only once. If the knight ends on a square that is one knight's move from the beginning square (so that it could tour the board again immediately, following the same path), the tour is closed, otherwise it is open.

The knight's tour problem is the mathematical problem of finding a knight's tour. Creating a program to find a knight's tour is a common problem given to computer science students. Variations of the knight's tour problem involve chessboards of different sizes than the usual  $8 \times 8$ , as well as irregular (non-rectangular) boards.

### Number of tours

On an  $8 \times 8$  board, there are exactly 26,534,728,821,064 directed closed tours (i.e. two tours along the same path that travel in opposite directions are counted separately, as are rotations and reflections. The number of undirected closed tours is half this number, since every tour can be traced in reverse. There are 9,862 undirected closed tours on a  $6 \times 6$  board.

But in our project, we will be showing only one of the feasible solution to the problem.

---

## INTRODUCTION

Chess is a two player game played on square board. A knight's tour is a sequence of squares such that all pairs of consecutive squares are adjacent in a graph and no square appears in the sequence more than once. In general, it is a Hamiltonian path problem with nodes as squares of the chessboard and the nodes are adjacent to each other only if a knight can move to the other node in single step. In this problem, we traverse the graph such that every nodes are visited exactly once and no nodes are left unvisited. There can exist more than one such tours possible in a chess board for provided starting node. And this project finds a single solution out of many.

### **The Knight's tour problem | Backtracking**

Backtracking is an algorithmic paradigm that tries different solutions until finds a solution that "works". Problems which are typically solved using backtracking technique have following property in common. These problems can only be solved by trying every possible configuration and each configuration is tried only once. A Naive solution for these problems is to try all configurations and output a configuration that follows given problem constraints. Backtracking works in incremental way and is an optimization over the Naive solution where all possible configurations are generated and tried.

For example, consider the following Knight's Tour problem. The knight is placed on the first block of an empty board and, moving according to the rules of chess, must visit each square exactly once.

Following is chessboard with 8 x 8 cells. Numbers in cells indicate move number of Knight:

0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

## WORKING OF BACKTRACKING

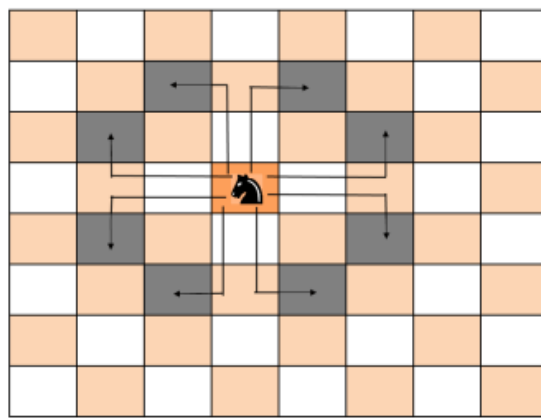
**Backtracking** works in an incremental way to attack problems. Typically, we start from an empty solution vector and one by one add items (Meaning of item varies from problem to problem. In context of Knight's tour problem, an item is a Knight's move). When we add an item, we check if adding the current item violates the problem constraint, if it does then we remove the item and try other alternatives. If none of the alternatives work out then we go to previous stage and remove the item added in the previous stage. If we reach the initial stage back then we say that no solution exists. If adding an item doesn't violate constraints then we recursively add items one by one. If the solution vector becomes complete then we print the solution.



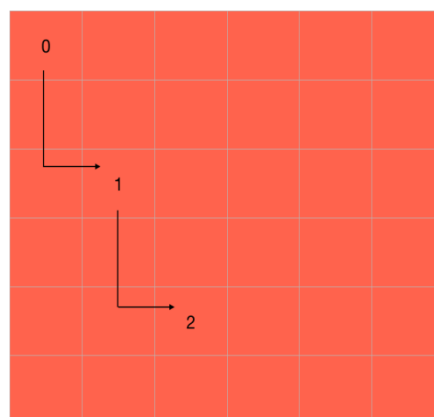
## **PROBLEM STATEMENT**

We start by moving the knight and if the knight reaches to a cell from where there is no further cell available to move and we have not reached to the solution yet (not all cells are covered), then we backtrack and change our decision and choose a different path.

So from a cell, we choose a move of the knight from all the moves available, and then recursively check if this will lead us to the solution or not. If not, then we choose a different path.



As you can see from the picture above, there is a maximum of 8 different moves which a knight can take from a cell. So if a knight is at the cell  $(i,j)$ , then the moves it can take are  $(i+2, j+1)$ ,  $(i+1, j+2)$ ,  $(i-2, j+1)$ ,  $(i-1, j+2)$ ,  $(i-1, j-2)$ ,  $(i-2, j-1)$ ,  $(i+1, j-2)$  and  $(i+2, j-1)$ .



We keep the track of the moves in a matrix. This matrix stores the step number in which we visited a cell. For example, if we visit a cell in the second step, it will have a value of 2.

1	30	47	52	5	28	43	54
48	51	2	29	44	53	6	27
31	46	49	4	25	8	55	42
50	3	32	45	56	41	26	7
33	62	15	20	9	24	39	58
16	19	34	61	40	57	10	23
63	14	17	36	21	12	59	38
18	35	64	13	60	37	22	11

This matrix also helps us to know whether we have covered all the cells or not. If the last visited cell has a value of  $N*N$ , it means we have covered all the cells.

Thus, our approach includes starting from a cell and then choosing a move from all the available moves. Then we check if this move will lead us to the solution or not. If not, we choose a different move. Also, we store all the steps in which we are moving in a matrix.

## **OBJECTIVE**

- To solve general Knight's tour problem.
- To implement possible heuristics to reduce the complexity while traversing.
- To practice on tree traversals, graph traversal dealing with real world problem.

A knight's tour is a sequence of moves of a knight on a chessboard such that the knight visits every square only once. If the knight ends on a square that is one knight's move from the beginning square (so that it could tour the board again immediately, following the same path), the tour is closed, otherwise it is open.

### **Approach:**

- Create a solution matrix of the same structure as chessboard.
- Start from 0,0 and index = 0. (index will represent the no of cells has been covered by the knight)
- Check current cell is not already used if not then mark that cell (start with 0 and keep incrementing it, it will show us the path for the knight).
- Check if index =  $N*N-1$ , means Knight has covered all the cells. return true and print the solution matrix.
  - Now try to solve rest of the problem recursively by making index +1. Check all 8 directions. (Knight can move to 8 cells from its current position.) Check the boundary conditions as well
  - If none of the 8 recursive calls return true, BACKTRACK and undo the changes ( put 0 to corresponding cell in solution matrix) and return false.
- See the code for better understanding.

## **METHODOLOGY**

### **ALGORITHM**

#### **1. Naive Algorithm for Knight's tour**

The Naive Algorithm is to generate all tours one by one and check if the generated tour satisfies the constraints.

```
while there are untried tours
{
    generate the next tour
    if this tour covers all squares
    {
        print this path;
    }
}
```

#### **2. Backtracking Algorithm for Knight's tour**

Following is the Backtracking algorithm for Knight's tour problem.

If all squares are visited

    print the solution

Else

a) Add one of the next moves to solution vector and recursively check if this move leads to a solution. (A Knight can make maximum eight moves. We choose one of the 8 moves in this step).

b) If the move chosen in the above step doesn't lead to a solution then remove this move from the solution vector and try other alternative moves.

c) If none of the alternatives work then return false (Returning false will remove the previously added item in recursion and if false is returned by the initial call of recursion then "no solution exists" )

```

KNIGHT-TOUR(sol, i, j, step_count, x_move, y_move)

    if step_count == N*N
        return TRUE

    for k in 1 to 8
        next_i = i+x_move[k]
        next_j = j+y_move[k]

        if IS-VALID(next_i, next_j, sol)
            sol[next_i][next_j] = step_count

            if KNIGHT-TOUR(sol, next_i, next_j, step_count+1, x_move, y_move)
                return TRUE

            sol[next_i, next_j] = -1

    return FALSE

START-KNIGHT-TOUR()

    sol = [[]]

    for i in 1 to N
        for j in 1 to N
            sol[i][j] = -1

    x_move = [2, 1, -1, -2, -2, -1, 1, 2]
    y_move = [1, 2, 2, 1, -1, -2, -2, -1]

    sol[1, 1] = 0

    if KNIGHT-TOUR(sol, 1, 1, 1, x_move, y_move)
        return TRUE

    return FALSE

```

## **SOURCE CODE:**

//program to solve knight's tour problem using backtracking.

```
#include<stdio.h>
#include <stdbool.h>
#define N 8

int solveKTUtil(int x, int y, int movei, int sol[N][N], int xMove[],
               int yMove[]);

/*function to check if i,j are valid indexes for N*N chessboard */
int isSafe(int x, int y, int sol[N][N])
{
    if ( x >= 0 && x < N && y >= 0 && y < N && sol[x][y] == -1)
        return 1;
    return 0;
}

/* function to print solution matrix sol[N][N] */
void printSolution(int sol[N][N])
{
    for (int x = 0; x < N; x++)
    {
        for (int y = 0; y < N; y++)
            printf(" %2d ", sol[x][y]);
        printf("\n");
    }
}

/* This function solves the Knight Tour problem using Backtracking. This
function mainly uses solveKTUtil() to solve the problem. */
bool solveKT()
{
    int sol[N][N];

    /* Initialization of solution matrix */
    for (int x = 0; x < N; x++)
        for (int y = 0; y < N; y++)
            sol[x][y] = -1;

    /* xMove[] and yMove[] define next move of Knight.
    xMove[] is for next value of x coordinate
    yMove[] is for next value of y coordinate */
    int xMove[8] = { 2, 1, -1, -2, -2, -1, 1, 2 };
    int yMove[8] = { 1, 2, 2, 1, -1, -2, -2, -1 };
```

```

// Since the Knight is initially at the first block
sol[0][0] = 0;

/* Start from 0,0 and explore all tours using solveKTUtil() */
if(solveKTUtil(0, 0, 1, sol, xMove, yMove) == false)
{
    printf("Solution does not exist");
    return false;
}
else
    printSolution(sol);

return true;
}

/* A recursive utility function to solve Knight Tour problem */
int solveKTUtil(int x, int y, int movei, int sol[N][N], int xMove[N],
               int yMove[N])
{
    int k, next_x, next_y;
    if (movei == N*N)
        return true;

    /* Try all next moves from the current coordinate x, y */
    for (k = 0; k < 8; k++)
    {
        next_x = x + xMove[k];
        next_y = y + yMove[k];
        if (isSafe(next_x, next_y, sol))
        {
            sol[next_x][next_y] = movei;
            if (solveKTUtil(next_x, next_y, movei+1, sol, xMove, yMove) == true)
                return true;
            else
                sol[next_x][next_y] = -1;    // backtracking
        }
    }
    return false;
}

/* Driver program to test above functions */
int main()
{
    solveKT();
    getchar();
    return 0;
}

```

## **Time complexity**

Analysis of Code :

We are not going into the full time complexity of the algorithm. Instead, we are going to see how algorithm is efficient as compared to naive algorithm.

There are  $N*N$  i.e.,  $N^2$  cells in the board and we have a maximum of 8 choices to make from a cell, so we can write the worst case running time as  $O(8N^2)O(8N^2)$ .

But we don't have 8 choices for each cell. For example, from the first cell, we only have two choices.

Even considering this, our running time will be reduced by a factor and will become  $O(k^{N^2})$  instead of  $O(8^{N^2})$ .



## Gui to the knight's tour problem:

```
//Using warnsdorff's rule
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

typedef unsigned char cell;
int dx[] = { -2, -2, -1, 1, 2, 2, 1, -1 };
int dy[] = { -1, 1, 2, 2, 1, -1, -2, -2 };

void init_board(int w, int h, cell **a, cell **b)
{
    int i, j, k, x, y, p = w + 4, q = h + 4;
    /* b is board; a is board with 2 rows padded at each side */
    a[0] = (cell*)(a + q);
    b[0] = a[0] + 2;

    for (i = 1; i < q; i++) {
        a[i] = a[i-1] + p;
        b[i] = a[i] + 2;
    }

    memset(a[0], 255, p * q);
    for (i = 0; i < h; i++) {
        for (j = 0; j < w; j++) {
            for (k = 0; k < 8; k++) {
                x = j + dx[k], y = i + dy[k];
                if (b[i+2][j] == 255) b[i+2][j] = 0;
                b[i+2][j] += x >= 0 && x < w && y >= 0 && y < h;
            }
        }
    }
}

#define E "\033["
int walk_board(int w, int h, int x, int y, cell **b)
{
    int i, nx, ny, least;
    int steps = 0;
    printf(E"H"E"J"E"%d;%dH"E"32m[]"E"m", y + 1, 1 + 2 * x);

    while (1) {
        /* occupy cell */
        b[y][x] = 255;

        /* reduce all neighbors' neighbor count */
        for (i = 0; i < 8; i++)
            b[ y + dy[i] ][ x + dx[i] ]--;

        /* find neighbor with lowest neighbor count */
        least = 255;
        for (i = 0; i < 8; i++) {
```

```

        if (b[ y + dy[i] ][ x + dx[i] ] < least) {
            nx = x + dx[i];
            ny = y + dy[i];
            least = b[ny][nx];
        }
    }

    if (least > 7) {
        printf(E"%dH", h + 2);
        return steps == w * h - 1;
    }

    if (steps++) printf(E"%d;%dH[]", y + 1, 1 + 2 * x);
    x = nx, y = ny;
    printf(E"%d;%dH"E"31m[]"E"m", y + 1, 1 + 2 * x);
    fflush(stdout);
    usleep(320000);
}
}

int solve(int w, int h)
{
    int x = 0, y = 0;
    cell **a, **b;
    a = malloc((w + 4) * (h + 4) + sizeof(cell*) * (h + 4));
    b = malloc((h + 4) * sizeof(cell*));

    while (1) {
        init_board(w, h, a, b);
        if (walk_board(w, h, x, y, b + 2)) {
            printf("Success!\n");
            return 1;
        }
        if (++x >= w) x = 0, y++;
        if (y >= h) {
            printf("Failed to find a solution\n");
            return 0;
        }
        printf("Any key to try next start position");
        getchar();
    }
}

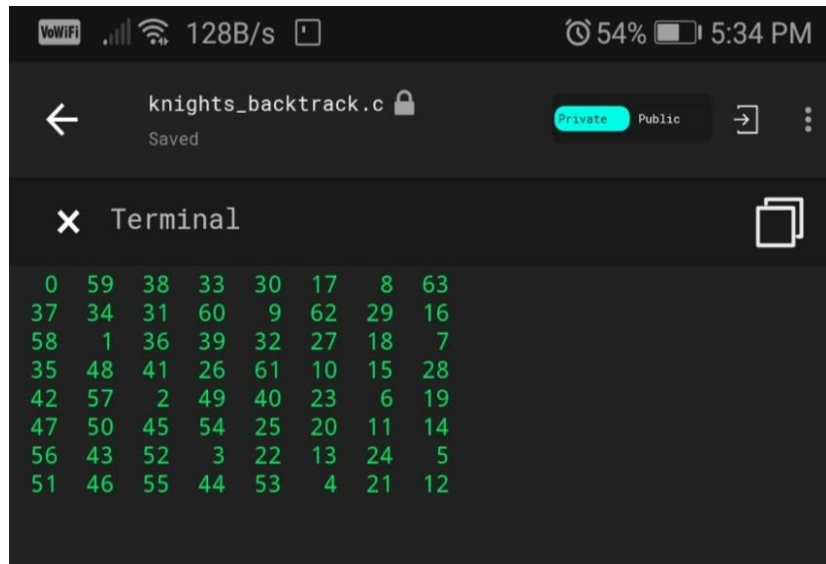
int main(int c, char **v)
{
    int w, h;
    if (c < 2 || (w = atoi(v[1])) <= 0) w = 8;
    if (c < 3 || (h = atoi(v[2])) <= 0) h = w;
    solve(w, h);

    return 0;
}

```

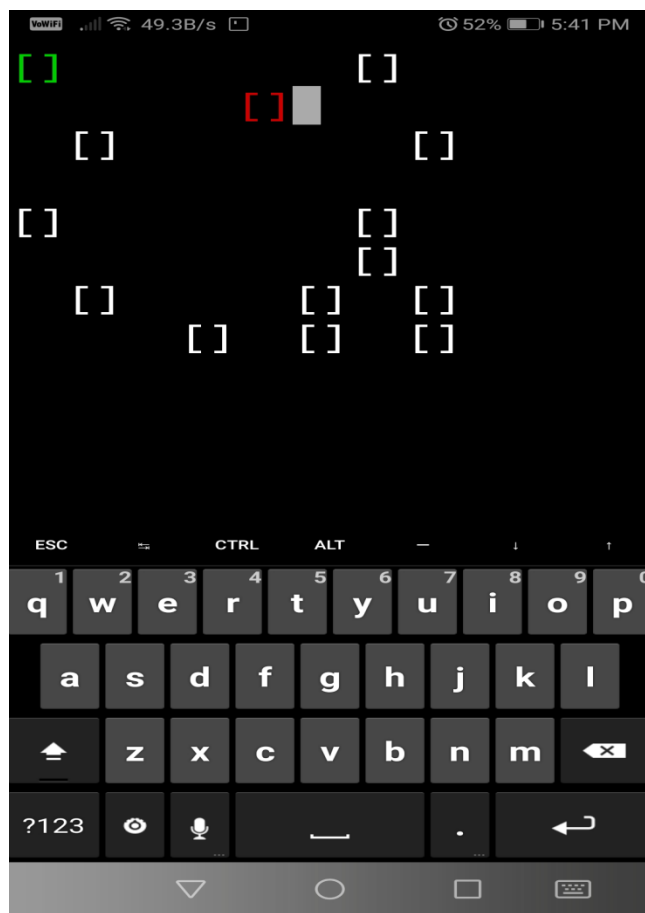
## RESULTS /OUTPUTS

### OUTPUT OF KNIGHT'S TOUR WITHOUT GUI



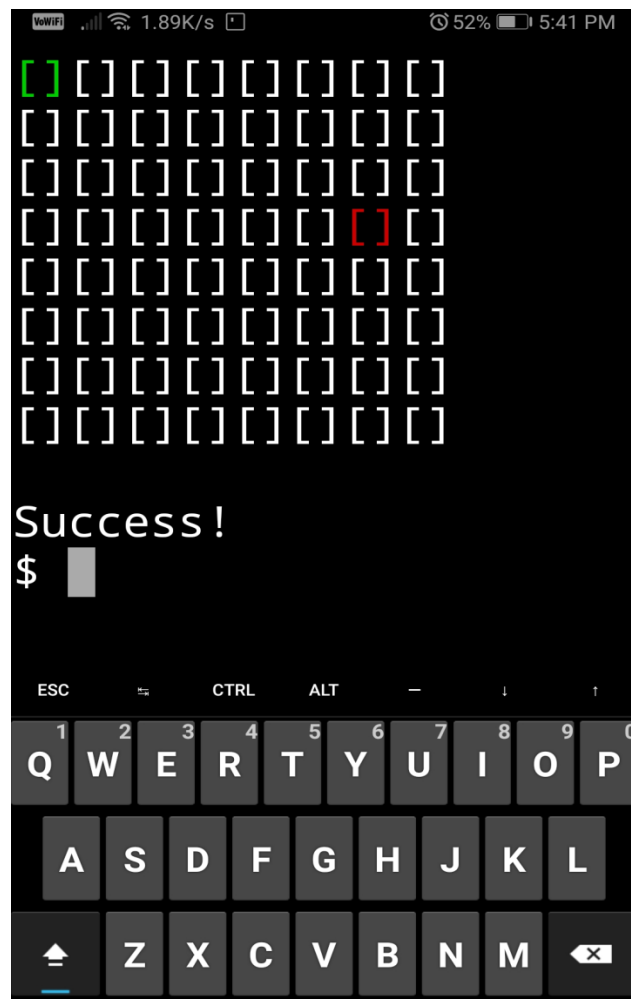
```
0 59 38 33 30 17 8 63
37 34 31 60 9 62 29 16
58 1 36 39 32 27 18 7
35 48 41 26 61 10 15 28
42 57 2 49 40 23 6 19
47 50 45 54 25 20 11 14
56 43 52 3 22 13 24 5
51 46 55 44 53 4 21 12
```

### OUTPUT OF KNIGHT'S TOUR WITH GUI



THIS IS THE ONGOING OUTPUT  
WHEN THE PROBLEM STILL  
EXECUTING.

Initial position: [0][0]



Final position: [3][6]

This is the final output with a success knight's tour by traversing each block exactly and only once.

### Output based on different start points of the knight



Initial position:[7][7]

Initial position:[4][5]

Final position:[1][6]

Final position:[1][5]

## **CONCLUSION**

By the end of this project we have learnt about backtracking that is used to solve any problem by taking one item at a time. When we add an item, we check if adding the current item violates the problem constraint, if it does then we remove the item and try other alternatives. If none of the alternatives work out then we go to previous stage and remove the item added in the previous stage .And this process continues till it reaches final state.

This algorithm is faster as compared to brute force approach as the time complexity is less compared to it.

Limitations of backtracking over Warnsdorff's rule:

Warnsdorff's rule is the best and efficient method to solve a knight's tour. Proficient usage of Data structures and the user interface help us to code and understand the tour easily.'

Warnsdorff's rule gives always a closed tour. '

## **REFERENCE**

- [https://en.wikipedia.org/wiki/Knight%27s\\_tour](https://en.wikipedia.org/wiki/Knight%27s_tour)
- <https://www.codesdope.com/course/algorithms-knights-tour-problem/>
- <https://www.geeksforgeeks.org/the-knights-tour-problem-backtracking-1/>
- <https://github.com/topics/knight-tour>
- [http://rosettacode.org/wiki/Knight's\\_Tour](http://rosettacode.org/wiki/Knight's_Tour)