COMPUTER AND INFORMATION ENGINEERING LAB I
ECIE 3101 (SECTION 1) / EECE 3312 (SECTION 1)
SEMESTER 2 2020/2021

PROJECT REPORT

Title: KNIGHT'S TOUR

Lecturer : Assoc. Prof. Dr. Rashidah Funke Olanrewaju
Group no. : 2
Group Members :

Name	Matric Number	Subject
MUHAMMAD AKMAL BIN JAINI	1815377	DSA/LAB
KHAIRUL IZHAM BIN KAMAL	1816045	LAB
NOR HALIZA NOR RAHIM	1826502	DSA/LAB
SARAH HANNANI BINTI ABDUL MANAH	1912850	LAB
ADRIANA NOH	1919540	DSA

1.0 Introduction

The Knight's Tour is usually played on a conventional chessboard, but it can also be played on any rectangular board with N rows and columns as well as irregular boards. The knight should search for a path from a starting point until it visits every square or exhaust all possibilities. It is a sequence of moves on a chessboard such that the knight visits every square exactly once. A knight can move in a special manner which is either two squares horizontally and one square vertically or two squares vertically and one square horizontally in each direction. This problem can have multiple solutions.

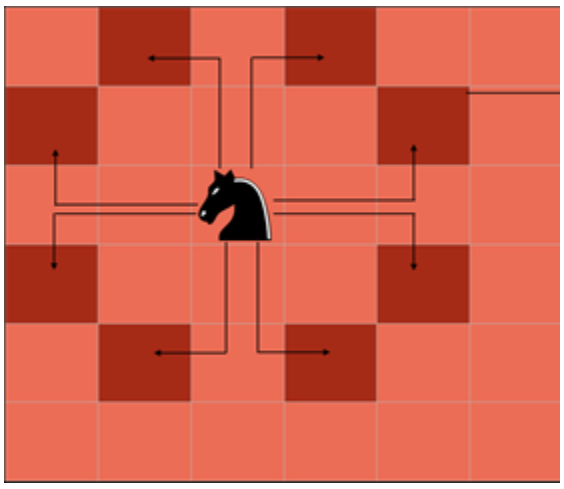

Figure 1 : shows the valid moves that the knight can move from that position.

Knight tour can be represented as a graph where all the squares on the board as vertices and all legal movements as edges. Size of the board is n-by-n, where n is an integer number. A total number of vertices is simply given by n^2 while the total number of edges is given by 4(n-2)(n-1). For an 8-by-8 chessboard, it contains 64 vertices and 168 edges. In order to approach the solution, a depth first search with backtracking method is implemented to make sure the knight goes to the square with the fewest number of possible following movements. The goal is that towards the end of the tour, it will have visited squares with more move options. It is efficient to visit nearby squares first so it can reach all vertices at the end of the path taken.

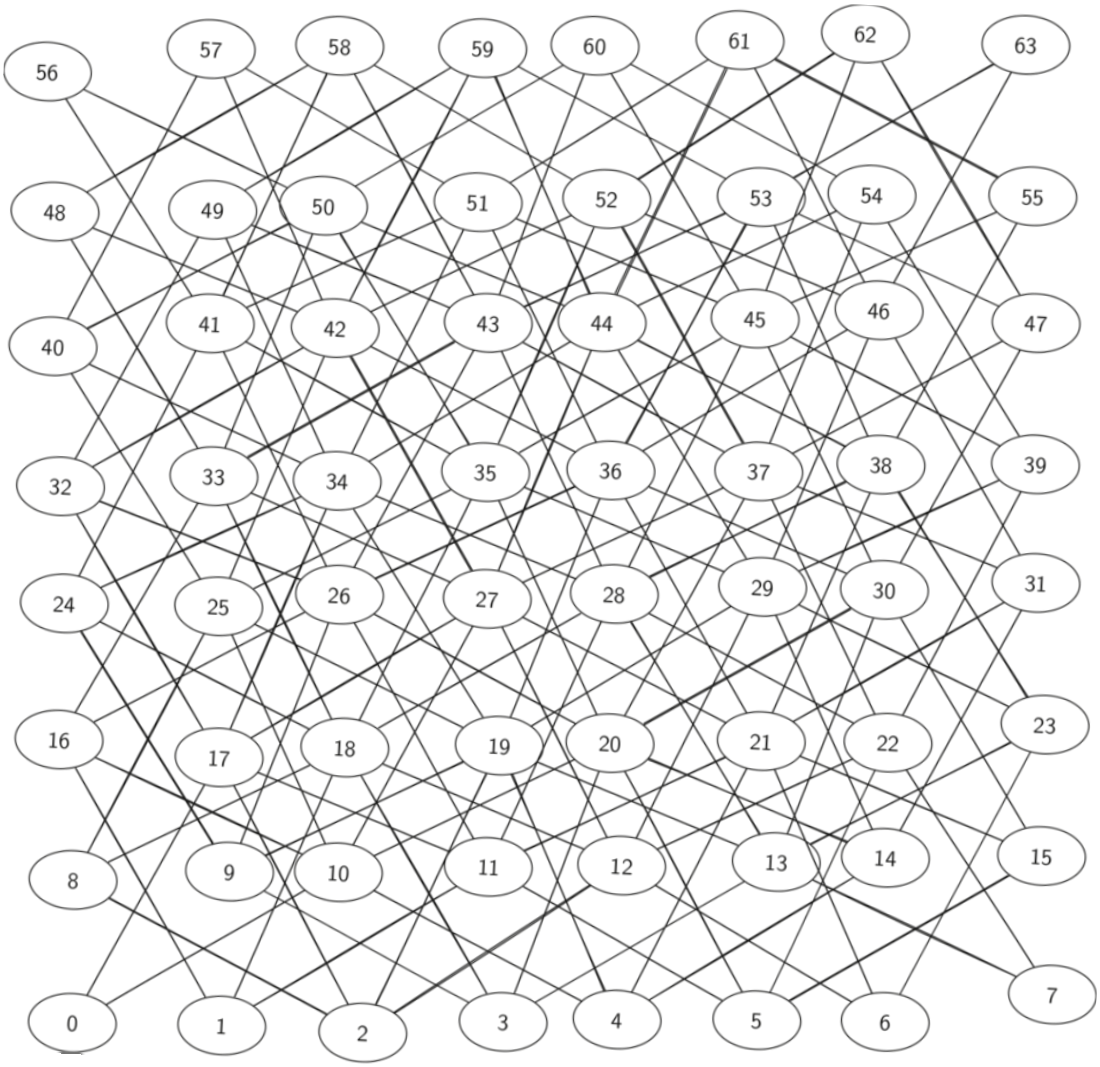

Figure 2 : shows a graph of 8-by-8 chessboard.

2.0 Source Code

```
import sys


class KnightsTour:
    def __init__(self, width, height):
        self.w = width
        self.h = height

        self.flag = False

        self.board = []
        self.generate_board()

    def path_length (self):
        print ("\nPath length is : ", counter)

    def path_existance (self):
        if exist == True:
            pass
        else:
            print ("\n ====== Path not found ======")

    def generate_board(self):
        """
        Creates a nested list to represent the game board
        """
        for i in range(self.h):
            self.board.append([0]*self.w)

    def print_board(self):
        print ("  ")
        print ("Knight Tour on ", self.h, " X ", self.w, " board :")
        print ("----------------------")
        for elem in self.board:
            print (elem)
        print ("----------------------")
        print ("  ")
```

```
def genLegalMoves(self, cur_pos):
    """
    Generates a list of legal moves for the knight to take next
    """
    possible_pos = []
    move_offsets = [(1, 2), (1, -2), (-1, 2), (-1, -2),
                    (2, 1), (2, -1), (-2, 1), (-2, -1)]

    for move in move_offsets:
        new_x = cur_pos[0] + move[0]
        new_y = cur_pos[1] + move[1]

        if (new_x >= self.h):
            continue
        elif (new_x < 0):
            continue
        elif (new_y >= self.w):
            continue
        elif (new_y < 0):
            continue
        else:
            possible_pos.append((new_x, new_y))

    return possible_pos

def sort_lonely_neighbors(self, to_visit):
    """
    It is more efficient to visit the lonely neighbors first,
    since these are at the edges of the chessboard and cannot
    be reached easily if done later in the traversal
    """
    neighbor_list = self.genLegalMoves(to_visit)
    empty_neighbours = []

    for neighbor in neighbor_list:
        np value = self.board[neighbor[0]][neighbor[1]]
```

```
            if np_value == 0:
                empty_neighbours.append(neighbor)

        scores = []
        for empty in empty_neighbours:
            score = [empty, 0]
            moves = self.genLegalMoves(empty)
            for m in moves:
                if self.board[m[0]][m[1]] == 0:
                    score[1] += 1
            scores.append(score)

        scores_sort = sorted(scores, key = lambda s: s[1])
        sorted_neighbours = [s[0] for s in scores_sort]

        return sorted_neighbours
    def tour(self, n, path, to_visit):
        """
        Recursive definition of knights tour. Inputs are as follows:
        n = current depth of search tree
        path = current path taken
        to_visit = node to visit
        """
        self.board[to_visit[0]][to_visit[1]] = n
        path.append(to_visit) #append the newest vertex to the current point
        print ("Visiting: ", to_visit)
        global counter
        counter = counter + 1
        global exist
        exist = False
        repeat = self.w*self.h

        if n == repeat: #if every grid is filled
            self.print board()
```

```
            print ("Path movement in coordinate :")
            print (path)
            print ("----------------------")
            self.path_length()
            print ("\n====== Done! ======")
            exist = True

        else:
            sorted_neighbours = self.sort_lonely_neighbors(to_visit)
            for neighbor in sorted_neighbours:
                if repeat == counter:
                    break
                self.tour(n+1, path, neighbor)

            #If we exit this loop, all neighbours failed so we reset
            if sorted_neighbours != 0 and counter != repeat and n!=1:
                self.board[to_visit[0]][to_visit[1]] = 0
                path.pop()
                print ("Going back to : ", path[-1])

            elif sorted_neighbours != 0 and counter == repeat and n==1:
                self.path_existance()



if __name__ == '__main__':
    #Define the size of grid by inputing value of n
    while True:
        while True:
            n = input("\nPlease enter a positive integer for grid size n x n : ")
            try:
                value = int(n)
                if value > 0 :
                    print("Input is a positive integer :", value,"\n")
                    break;
```

```
        else :
            print("!!! THIS IS NEGATIVE INTEGER !!! : ", value,"\n")
    except ValueError:
        try:
            float(n)
            print("!!! THIS IS FLOAT NUMBER !!! ", n, "\n")
        except ValueError:
            print("!!! THIS IS NOT A NUMBER !!! \n")
n = value
counter = 0
kt = KnightsTour( n, n)
kt.tour(1, [], (0,0))
print ("\n==================================")
print ("\nDo you wish to enter other value for grid size?")
print ("Please enter :")
print ("\n Y for Yes \n N for No")

while True:
    answer = input("\nAnswer : ")
    if answer == 'y' or answer == 'Y' :
        break
    elif answer == 'n' or answer == 'N' :
        sys.exit(1)
    else:
        print (" Please enter valid answer only ")
```

3.0 Results

```
Please enter a positive integer for grid size n x n : 8
Input is a positive integer : 8

Visiting:  (0, 0)
Visiting:  (1, 2)
Visiting:  (2, 0)
Visiting:  (0, 1)
Visiting:  (1, 3)
Visiting:  (0, 5)
Visiting:  (1, 7)
Visiting:  (3, 6)
Visiting:  (5, 7)
Visiting:  (7, 6)
Visiting:  (6, 4)
Visiting:  (7, 2)
Visiting:  (6, 0)
Visiting:  (4, 1)
Visiting:  (6, 2)
Visiting:  (7, 0)
Visiting:  (5, 1)
Visiting:  (3, 0)
Visiting:  (1, 1)
Visiting:  (0, 3)
Visiting:  (1, 5)
Visiting:  (0, 7)
Visiting:  (2, 6)
Visiting:  (4, 7)
Visiting:  (6, 6)
Visiting:  (7, 4)
Visiting:  (5, 5)
Visiting:  (6, 7)
Visiting:  (7, 5)
Visiting:  (6, 3)
Visiting:  (7, 1)
Visiting:  (5, 0)
Visiting:  (3, 1)
Visiting:  (1, 0)
```

```
Visiting:  (2, 2)
Visiting:  (4, 3)
Visiting:  (2, 4)
Visiting:  (3, 2)
Visiting:  (4, 0)
Visiting:  (5, 2)
Visiting:  (7, 3)
Visiting:  (6, 1)
Visiting:  (5, 3)
Visiting:  (4, 5)
Visiting:  (3, 7)
Visiting:  (1, 6)
Visiting:  (0, 4)
Visiting:  (2, 5)
Visiting:  (0, 6)
Visiting:  (2, 7)
Visiting:  (4, 6)
Visiting:  (3, 4)
Visiting:  (4, 2)
Visiting:  (5, 4)
Visiting:  (3, 3)
Visiting:  (2, 1)
Visiting:  (0, 2)
Visiting:  (1, 4)
Visiting:  (3, 5)
Visiting:  (2, 3)
Visiting:  (4, 4)
Visiting:  (5, 6)
Visiting:  (7, 7)
Visiting:  (6, 5)
```

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

Path movement in coordinate :
[(0, 0), (1, 2), (2, 0), (0, 1), (1, 3), (0, 5), (1, 7), (3, 6), (5, 7), (7, 6), (6, 4), (7, 2), (6, 0), (4, 1), (6,
2), (7, 0), (5, 1), (3, 0), (1, 1), (0, 3), (1, 5), (0, 7), (2, 6), (4, 7), (6, 6), (7, 4), (5, 5), (6, 7), (7, 5), (
6, 3), (7, 1), (5, 0), (3, 1), (1, 0), (2, 2), (4, 3), (2, 4), (3, 2), (4, 0), (5, 2), (7, 3), (6, 1), (5, 3), (4, 5)
, (3, 7), (1, 6), (0, 4), (2, 5), (0, 6), (2, 7), (4, 6), (3, 4), (4, 2), (5, 4), (3, 3), (2, 1), (0, 2), (1, 4), (3,
 5), (2, 3), (4, 4), (5, 6), (7, 7), (6, 5)]
---------------------

Path length is :  64

====== Done! ======

====================================

Do you wish to enter other value for grid size?
Please enter :

 Y for Yes
 N for No

Answer :
```

```
Please enter a positive integer for grid size n x n : 5
Input is a positive integer : 5

Visiting:  (0, 0)
Visiting:  (1, 2)
Visiting:  (0, 4)
Visiting:  (2, 3)
Visiting:  (4, 4)
Visiting:  (3, 2)
Visiting:  (4, 0)
Visiting:  (2, 1)
Visiting:  (3, 3)
Visiting:  (4, 1)
Visiting:  (2, 0)
Visiting:  (0, 1)
Visiting:  (1, 3)
Visiting:  (3, 4)
Visiting:  (4, 2)
Visiting:  (3, 0)
Visiting:  (1, 1)
Visiting:  (0, 3)
Visiting:  (2, 4)
Visiting:  (4, 3)
Visiting:  (3, 1)
Visiting:  (1, 0)
Visiting:  (2, 2)
Visiting:  (1, 4)
Visiting:  (0, 2)
```

```
Knight Tour on  5  X  5  board :
---------------------
[1, 12, 25, 18, 3]
[22, 17, 2, 13, 24]
[11, 8, 23, 4, 19]
[16, 21, 6, 9, 14]
[7, 10, 15, 20, 5]
---------------------

Path movement in coordinate :
[(0, 0), (1, 2), (0, 4), (2, 3), (4, 4), (3, 2), (4, 0), (2, 1), (3, 3), (4, 1), (2, 0), (0, 1), (1, 3), (3, 4), (4,
2), (3, 0), (1, 1), (0, 3), (2, 4), (4, 3), (3, 1), (1, 0), (2, 2), (1, 4), (0, 2)]
---------------------

Path length is :  25

====== Done! ======

==================================

Do you wish to enter other value for grid size?
Please enter :

 Y for Yes
 N for No

Answer :
```

4.0 Discussion

This program applies the concept of graph search algorithm depth first search (DFS) to solve the knight's tour problem. Interesting fact about knight tour is that it's a cyclic bipartite graph in which no two graph vertices within the same set are adjacent.

Starting off the source code with introducing the class `KnightsTours`. The first constructor `__init__` is defined with three parameters which are self, width and height. Constructors `path_length` to print the length and `path_existance` to check if there is a path are also introduced. Another constructor `generate_board` with self parameter is also initialised to create a nested list to represent the game board of the Knight's Tour with a condition for `i` in `height` range. Constructor `print_board` is defined to print the elements in the board. `genLegalMoves` is to create a list of legal moves. Here, all possible moves are listed. As mentioned, a knight can only move in an L-shape which means that it moves two squares vertically in any direction (2) and then one square horizontally (1) or two squares horizontally (2) and then one square vertically (1). Assuming it starts at [0, 0], it can move in 8 different combinations which are (1, 2), (1, -2), (-1, 2), (-1, -2), (2, 1), (2, -1), (-2, 1) and (-2, -1) as proposed under the function `genLegalMoves`. It takes the position of the knight on the board and generates all 8 possible moves.

Generally, a graph for an 8-by-8 chessboard contains 168 edges which can be seen in Figure 2. These edges are referring to all legal moves that can be made by the knight from a square. Since there are many vertices that can be chosen, a function called `sort_lonely_neighbors` was introduced to decide which nearby square needs to choose in the next step. `sort_lonely_neighbors` with self and `to_visit` parameters to visit the lonely neighbors first since these are at the edges of the chessboard and are hard to be reached if done later in the traversal.

The next important function is called `tour`. It's a recursive function that visits any possible nodes or squares and determines whether the path for knight tour exists or not. It has four parameters which are self, `n` which is the current depth in the search tree, `path` which is a list of the current path taken and `to_visit` which is the node to be visited. In this function, there are two global variables which are called `counter` and `exist.` `Counter` is to calculate how many nodes have been visited by the knight without turning back to the previous node while

`exist` is a boolean variable which has been set as False until path is found, it will change to True. In general, a knight's tour exists when the size of the board is 5-by-5 and above. However, if the condition of board size is not met, the function will still run until it stops when no possible legal moves can be made. All vertices visited are listed and printed and this part is looped until every grid is filled.

The next part in the source code shows how the user is asked to input a positive integer and the value is assigned to `n`. The value of `n` is then assigned to the width and height or the size of the board n-by-n. If the input is a negative value, the user is asked to enter another positive integer. If the input is a positive value, the main function `tour` is called with value (n, n). The program shows all the visiting nodes and the Knight's Tour on an n-by-n board is printed. The path movement is also shown in the form of coordinates and the path length is counted. The user is then asked if they wish to enter another value to prompt a new n-by-n Knight's Tour and if the input is 'y' or 'Y', the whole function is repeated again. If the input is 'n' or 'N', the program is terminated with a sys.exit(1) function.

5.0 Conclusion

The Knight's Tour problem has been around for centuries and has been investigated by famous mathematicians including Leonhard Euler. The heuristic rule proposed by the mathematician H.C. Warnsdorff has been very successful in generating numerous tours. From the general understanding of Warnsdorff's rule, the knight is moved so that it always proceeds to the square from which the knight will have the fewest onward moves. When calculating the number of onward moves for each candidate square, we do not count moves that revisit any square already visited. Warnsdorff's rule algorithm is the best and efficient method to solve a knight's tour since it always gives a closed tour. This particular heuristic helps humans to make decisions and heuristic searches that are often used in the field of artificial intelligence. Through proficient usage of data structures implemented in this project, this helps us to code and understand the Knight's Tour problem better.