KNIGHT TOUR PROBLEM

Cluster Innovation Centre

University of Delhi

1. **Anshika (11909)**
2. **Bhavya Tewari (11913)**
3. **Bhavya Verma (11914)**
4. **Gaurav Dubey (11919)**

Apr 2020

Month Long Project submitted for the paper

II.2 Understanding real life situations through Discrete Mathematics

Certificate of Originality

The work embodied in this report entitled **“Knight Tour Problem”** has been carried out by **Anshika, Bhavya Tewari, Bhavya Verma, Gaurav Dubey**for the paper “***II.2 Understanding real life situations through Discrete Mathematics*”**. I declare that the work and language included in this project report is free from any kind of plagiarism.

INDEX

1. ABSTRACT

The Knight Tour Problem is a classical problem in the game of Chess. The idea is to make a knight piece trace each square in a n x n Chess board (conventionally n=8). The knight is supposed to trace with its defined move i.e., two squares away horizontally and one square vertically, somewhat trailing an **L-** shape.

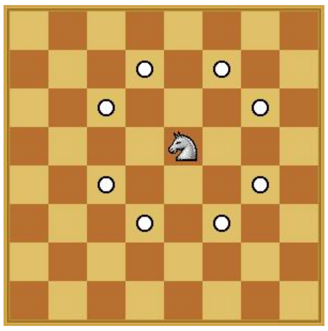


Fig. Knight’s Movement

The project aims to solve this problem using the idea of tracing paths and circuit through a graph with a given set of vertices and edges and define an algorithm for the same.

2. INTRODUCTION

2.1. Background

Some concepts from the graph theory are essential to arrive at the solution to a solution of the Knight's Tour problem.

Paths and Cycles

A walk is a traversal, in which the edges and vertices can be repeated. If all the vertices are distinct in a walk, we get a **path.** If the path terminates at the source vertex, it is known as a **cycle.**

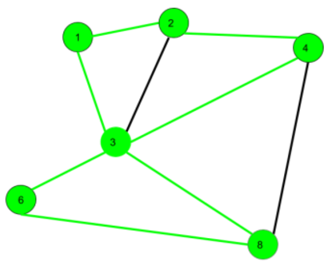


Fig 2.1.1: An example of a cycle.

Note: If we don't consider the edge (3,4) the walk will be known as path only, starting at 3 and terminating at 8. Similarly, other edge can be removed from the cycle to get path.

Connected Graphs

A connected graph is a graph in which it's possible to get from every vertex in the graph to every other vertex through a series of edges, called a path.

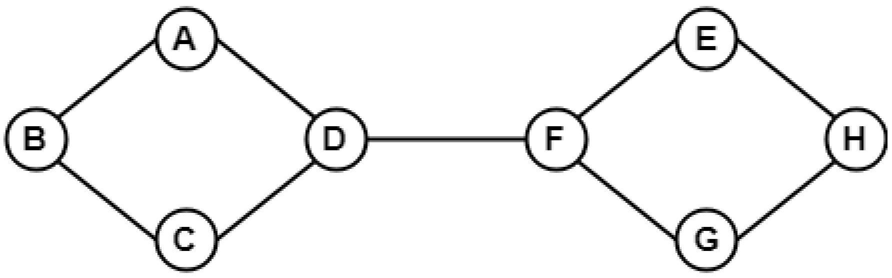


Fig1.1.2: An example of Connected Graph

Tours on a Graph

A path or a cycle may not give us the possible route which traverses through each edge of a given connected graph. Eulerian and Hamiltonian tours are two such traversals.

Eulerian Circuit

A circuit is a path in which the source and the terminal vertex is same. In Eulerian circuit we traverse through each edge of the graph.

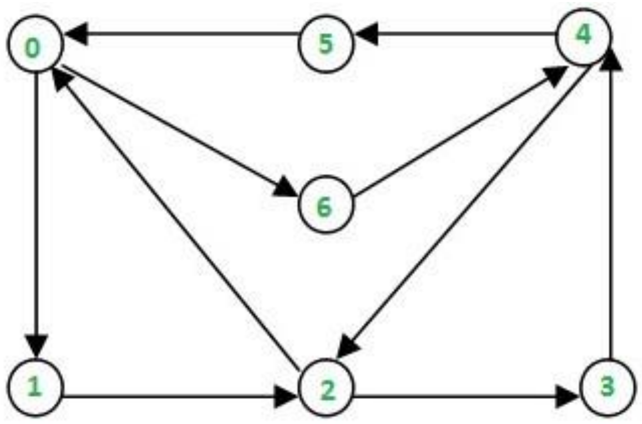


Fig 1.1.3: Eulerian Circuit- 0->6->5->4->0->1->2->3->4->2->0

Hamiltonian Cycle

Similar to Eulerian circuit, a Hamiltionian path is one which passes through each and every vertex without repeating the vertices.

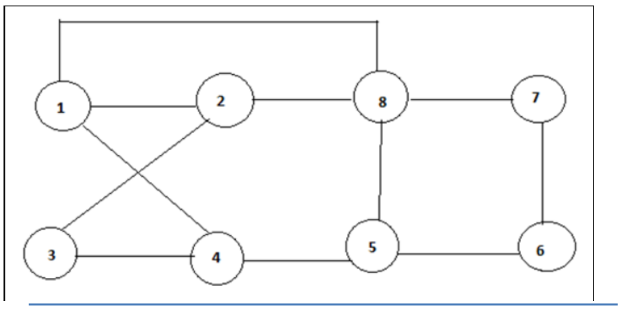


Fig 1.1.4: Hamiltonian Cycle: 1->2->3->4->5->6->7->8->1

2.2. Scopes and Objectives

The scope of the project is to use some key concepts of Graph theory to find the possibility for a knight piece to trace a route, visiting each square only once.

This has been done by developing an algorithm that inputs the dimension of the chess board and gives us a possible tour.

The final objective is to deeply understand the implementation of touring concepts of Graph theory (Hamiltonian cycle, Eulerian circuit) and relate them with several algorithmic paradigms.

The following section provides the methodology and resultant outputs of the same.

3. METHODOLOGY

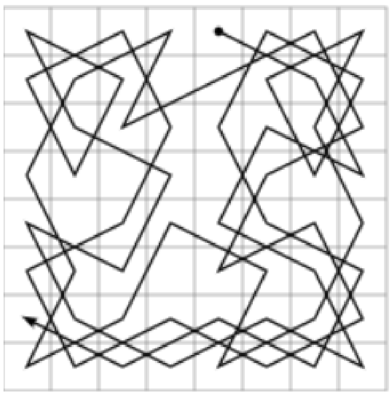


Fig3.1: Knight traversal in 8x8 chess board

The solution has been written in Java by implementing a backtracking algorithm.

Backtracking Algorithm

Backtracking is an algorithmic paradigm which tries different solutions until it arrives at a solution that works. All problems which can be solved through backtracking algorithms have a property in common that is i.e., the problem can only be solved by trying different configurations or using several data sets. A naïve solution is to try every configuration to finally arrive at the one that works.

The trial has to be done under given constraint which is also common property among such problems. In the knight tour, the validity of the co-ordinate we get after each movement is once such constraint, the other being visiting each square only once.

Before arriving at a definite backtracking algorithm we construct a naïve algorithm for Knight Tour.

```
While there are untraced tours {
        Trace a new route
        If, this route visits all squares only once{
        Print this path
}
}
```

The backtracking algorithm for the knight tour has been implemented in the method solveTraversal() in the class KnightTour which has been discussed in section 3.2.

The following is the backtracking algorithm for Knight Tour.

[1] If all squares have been visited

Print the solution matrix.

[2] Else

A. Append the possible moves to the current position and call isValidMove()function recursively is this move is a solution or not. (A knight is allowed 8 moves on the chess board, we find one of the 8 moves to be a solution)

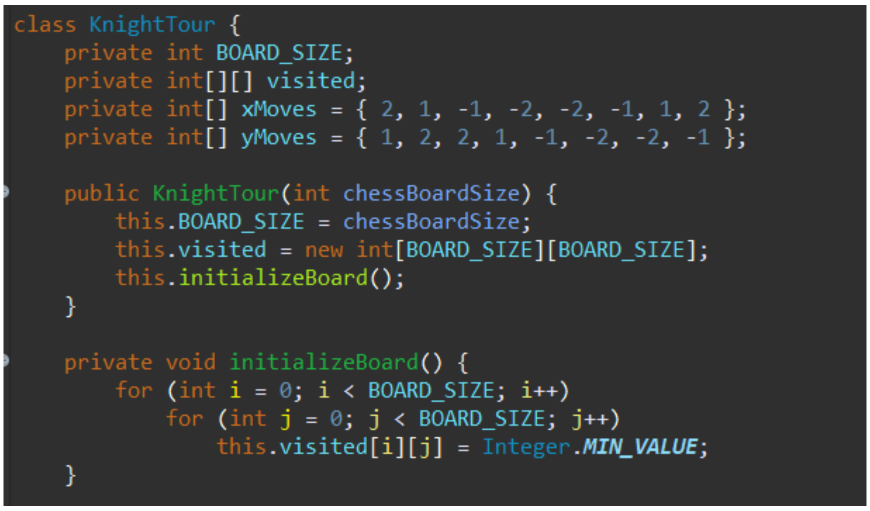
B. If the move does not happen to be a solution then try alternative moves.

C. Return false if no moves satisfy to be a part of the solution.

Returning false removes the previous moveCount in recursive call of the function and if false is returned initially for the given dimension then no feasible solution exists.

3.1. Movement and Boundary conditions

The class has two initialized one-dimensional correlated arrays x[] and y[] whose elements are either 1,-1,2,-2. The rationale behind the initialization of such an array comes from the defined movement of a knight on a chess board.



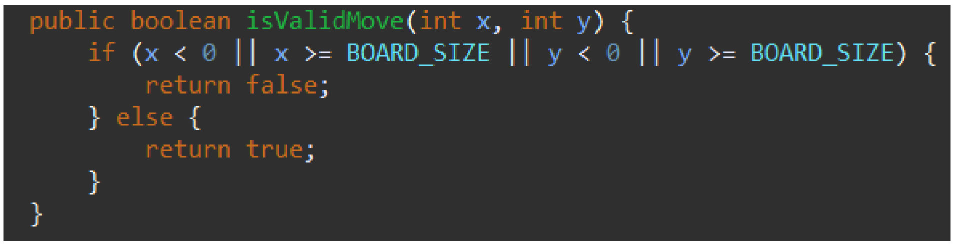
```
class KnightTour {
    private int BOARD_SIZE;
    private int[][] visited;
    private int[] xMoves = { 2, 1, -1, -2, -2, -1, 1, 2 };
    private int[] yMoves = { 1, 2, 2, 1, -1, -2, -2, -1 };

    public KnightTour(int chessBoardSize) {
        this.BOARD_SIZE = chessBoardSize;
        this.visited = new int[BOARD_SIZE][BOARD_SIZE];
        this.initializeBoard();
    }

    private void initializeBoard() {
        for (int i = 0; i < BOARD_SIZE; i++)
            for (int j = 0; j < BOARD_SIZE; j++)
                this.visited[i][j] = Integer.MIN_VALUE;
    }
```

Fig 3.1.1: Initialization of n x n matrix with negative value and arrays to change x and y co-ordinates

The function initializeBoard**()**initializes each element of the two dimensional matrix with a negative value, so as to later verify whether the next square the knight will be positioned at, has been visited or not.

Considering a knight on square of an8x8 chess board, we can say that the knight is active on a square with co-ordinates in the chess board (where, $0 \leq i, j \leq 7$). The knight is restricted to a movement of 2 spaces in one direction and 1 space in the other. This so there can be either a decrement or an increment in the x or y co-ordinate of the knight by 1 and 2. That is the next possible co-ordinates can be (i+1,j+2), (i+1,j-2), (i-1,j+2), (i-1,j-2), (i+2,j+1), (i+2,j-1), (i-2,j-1), (i-2,j+1).

After the knight visits a new position, we need to check whether resultant co-ordinate, the knight might arrive after operating the current position with the two arrays, will possibly exist or not. For this we have created a method with a Boolean return type (true or false) that checks the co-ordinates to be in-bound.

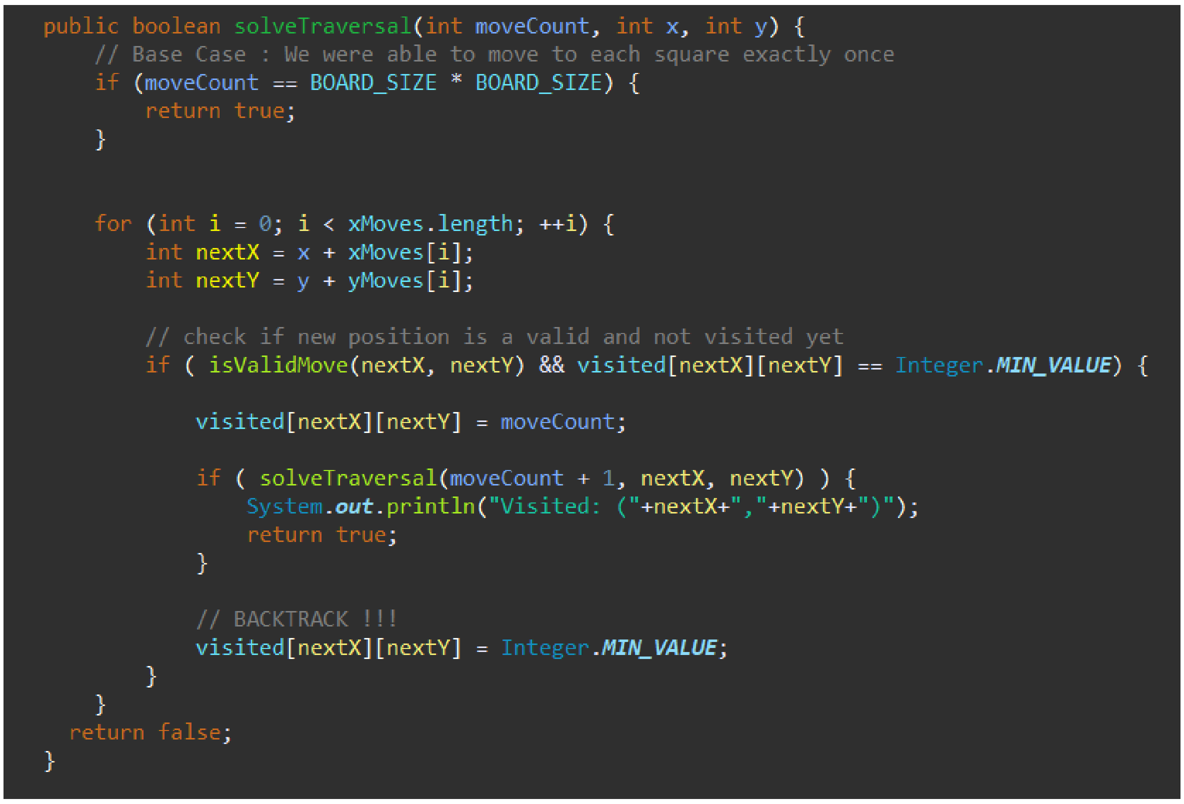


```
public boolean isValidMove(int x, int y) {
    if (x < 0 || x >= BOARD_SIZE || y < 0 || y >= BOARD_SIZE) {
        return false;
    } else {
        return true;
    }
}
```

Fig 3.1.2: Confirming validity of a resultant co-ordinate.

3.2. Traversal method

The traversal of the Knight through the chess board has been implemented by the method solveTraversal()which takes in the current position of the knight in the form of integer co-ordinates along with the moveCount whichis also the count of recursive implementations of the method. The integer variable moveCount ranges from 1 to 64 (or 1 to n*n).

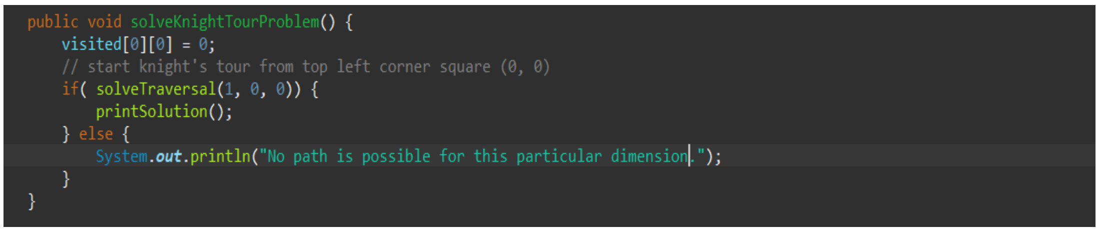


```
public boolean solveTraversal(int moveCount, int x, int y) {
    // Base Case : We were able to move to each square exactly once
    if (moveCount == BOARD_SIZE * BOARD_SIZE) {
        return true;
    }


    for (int i = 0; i < xMoves.length; ++i) {
        int nextX = x + xMoves[i];
        int nextY = y + yMoves[i];

        // check if new position is a valid and not visited yet
        if ( isValidMove(nextX, nextY) && visited[nextX][nextY] == Integer.MIN_VALUE) {

            visited[nextX][nextY] = moveCount;

            if ( solveTraversal(moveCount + 1, nextX, nextY) ) {
                System.out.println("Visited: ("+nextX+","+nextY+")");
                return true;
            }

            // BACKTRACK !!!
            visited[nextX][nextY] = Integer.MIN_VALUE;
        }
    }
  return false;
}
```

Fig 3.2.1: Traversal through the matrix.

The function processes the input parameters x and y by moving through the co-ordinates by 1,-1, 2 or -2 from xMoves[] and yMoves[] and appending them to get a new position. This new position is checked to be valid by the isValidMode() function. This function checks two parameters – that whether the co-ordinate is valid and the position has been visited or not (if not then the element will have a negative value). The element visited[nextX][nextY] is assigned the number of iterations on which it has been visited. The visited co-ordinate is also shown in the output.

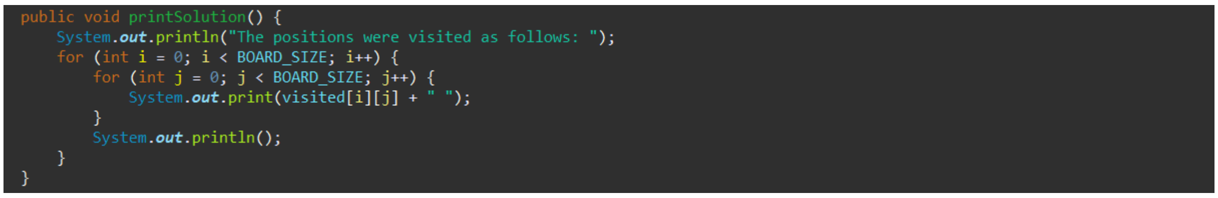
After a single iteration in the manner explained above, the function calls itself recursively and increments moveCount by 1 before processing it further.



```
public void solveKnightTourProblem() {
    visited[0][0] = 0;
    // start knight's tour from top left corner square (0, 0)
    if( solveTraversal(1, 0, 0)) {
        printSolution();
    } else {
        System.out.println("No path is possible for this particular dimension.");
    }
}
```

Fig 3.2.2: Calling traversal function.

The solveKnightTourProblem() method calls the solveTraversal() method with the initial position (0,0). The method also calls the solution to printSolution()method which prints a matrix in the output. If no solution is possiblefor a given dimension, a message will be printed for the same.



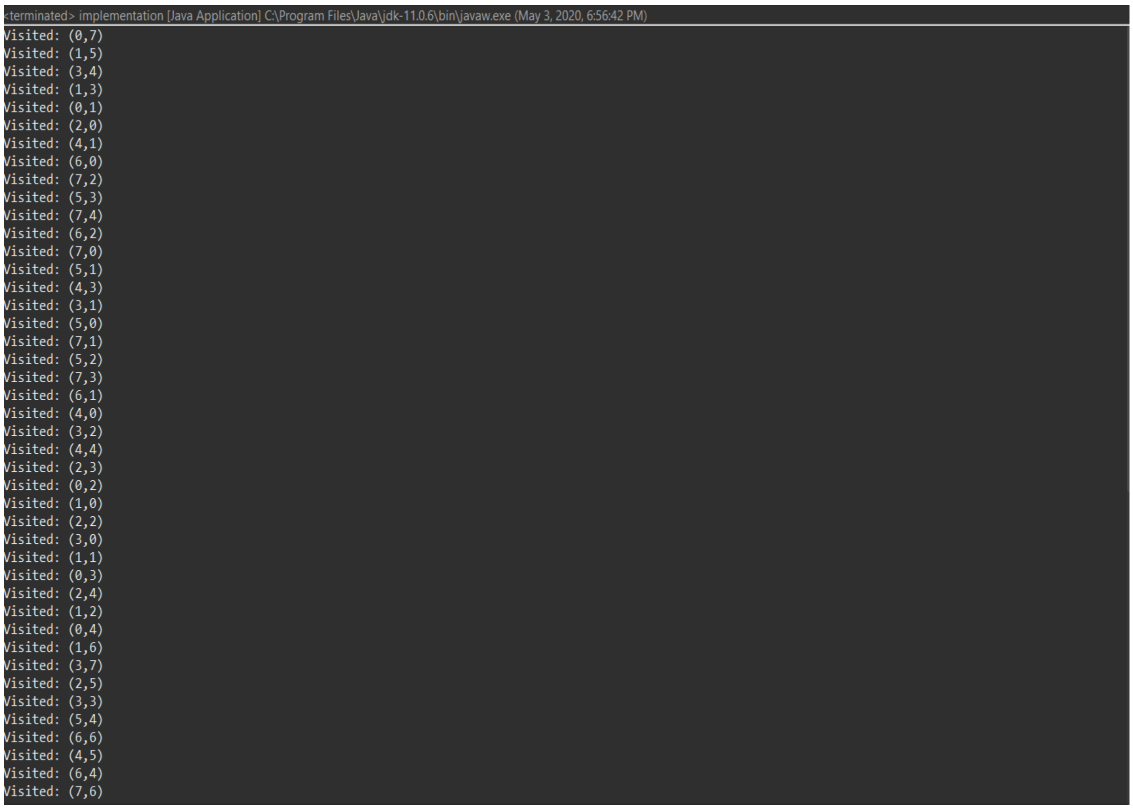
```
public void printSolution() {
    System.out.println("The positions were visited as follows: ");
    for (int i = 0; i < BOARD_SIZE; i++) {
        for (int j = 0; j < BOARD_SIZE; j++) {
            System.out.print(visited[i][j] + " ");
        }
        System.out.println();
    }
}
```

Fig 3.2.3: Matrix element has the count of iterations it took to reach at that particular position.

The method printSolution()prints the value of elements in visited[][]array.

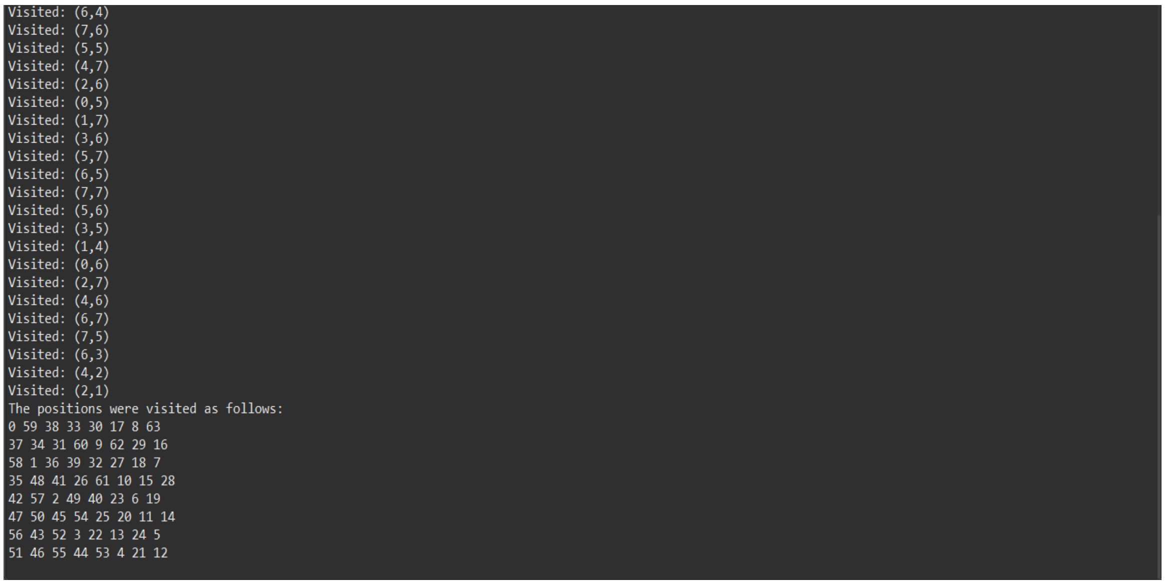
3.3. Output

The testcase was done for chess board of size 8x8. The co-ordinates of the squares traced by the knight is shown first and then a matrix is printed which shows the number of moves it took for the knight to reach at that particular position.



```
<terminated> implementation [Java Application] C:\Program Files\Java\jdk-11.0.6\bin\javaw.exe (May 3, 2020, 6:56:42 PM)
Visited: (0,7)
Visited: (1,5)
Visited: (3,4)
Visited: (1,3)
Visited: (0,1)
Visited: (2,0)
Visited: (4,1)
Visited: (6,0)
Visited: (7,2)
Visited: (5,3)
Visited: (7,4)
Visited: (6,2)
Visited: (7,0)
Visited: (5,1)
Visited: (4,3)
Visited: (3,1)
Visited: (5,0)
Visited: (7,1)
Visited: (5,2)
Visited: (7,3)
Visited: (6,1)
Visited: (4,0)
Visited: (3,2)
Visited: (4,4)
Visited: (2,3)
Visited: (0,2)
Visited: (1,0)
Visited: (2,2)
Visited: (3,0)
Visited: (1,1)
Visited: (0,3)
Visited: (2,4)
Visited: (1,2)
Visited: (0,4)
Visited: (1,6)
Visited: (3,7)
Visited: (2,5)
Visited: (3,3)
Visited: (5,4)
Visited: (6,6)
Visited: (4,5)
Visited: (6,4)
Visited: (7,6)
```

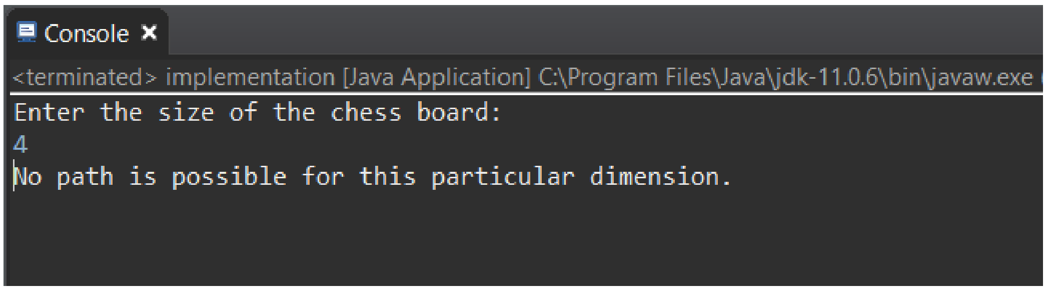
Fig 3.3.1(a): Coordinates of the positions visited.



```
Visited: (6,4)
Visited: (7,6)
Visited: (5,5)
Visited: (4,7)
Visited: (2,6)
Visited: (0,5)
Visited: (1,7)
Visited: (3,6)
Visited: (5,7)
Visited: (6,5)
Visited: (7,7)
Visited: (5,6)
Visited: (3,5)
Visited: (1,4)
Visited: (0,6)
Visited: (2,7)
Visited: (4,6)
Visited: (6,7)
Visited: (7,5)
Visited: (6,3)
Visited: (4,2)
Visited: (2,1)
The positions were visited as follows:
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
```

Fig 3.3.1(b): Matrix showing the count of iterations to reach a particular position.

No solution is feasible for n≤4. Thus if we change the dimension of the chess board we get the message for the same.



```
<terminated> implementation [Java Application] C:\Program Files\Java\jdk-11.0.6\bin\javaw.exe
Enter the size of the chess board:
4
No path is possible for this particular dimension.
```

Fig 3.3.2: No possible solutions for n≤4.

4. CONCLUSION

In the project, we were successfully able to find a possible solution of the Knight tour by implementing Backtracking algorithm. Another algorithm could be implemented by using the Warnsdorff's rule which states that the knight should be moved to a square which has minimal degree i.e., the least number of unvisited, adjacent squares. The starting position is essential for both implementations.

To get varied solutions to the tour, we can also have the starting position to be determined by the user. Although by implementing backtracking only once we can reach only at a single solution for a particular starting position. For getting all possible solutions for a particular starting position, the traversal function has to be called recursively for all possible moves from a particular solution at different iterations. This would take a more running time though. For instance,

There are 26,534,728,821,064 closed directed knight's tours and the number of undirected ones is half of that. If rotations and reflections are considered, then the number will increase by manifolds.

5. APPENDIX

Java implementation of the solution.

KnightTour.java (Contains all the methods which
are called in the test file

"implementation.java".)

```
package solution;

class KnightTour {

   private int BOARD_SIZE;

            private int[][] visited;

            private int[] xMoves = { 2, 1, -1, -2, -2, -1, 1, 2 };

            private int[] yMoves = { 1, 2, 2, 1, -1, -2, -2, -1 };

                        public KnightTour(int chessBoardSize) {

                        this.BOARD_SIZE = chessBoardSize;

                        this.visited = new int[BOARD_SIZE][BOARD_SIZE];

                        this.initializeBoard();

            }

            private void initializeBoard() {

                        for (int i = 0; i< BOARD_SIZE; i++)

                                    for (int j = 0; j < BOARD_SIZE; j++)

                                                this.visited[i][j] = Integer.MIN_VALUE;

            }

            public void solveKnightTourProblem() {

                        visited[0][0] = 0;

                        // start knight's tour from top left corner square (0, 0)

                        //Can be determined by the user to get a variety of soultions.

                        if( solveTraversal(1, 0, 0)) {

                                    printSolution();

                        } else {
```

```
                System.out.println("No path is possible for this particular dimension.");
        }
}

public booleansolveTraversal(int moveCount, int x, int y) {
        // Base Case : We were able to move to each square exactly once
        if (moveCount == BOARD_SIZE * BOARD_SIZE) {
                return true;
        }

        for (int i = 0; i<xMoves.length; ++i) {
                int nextX = x + xMoves[i];
                int nextY = y + yMoves[i];

                // check if new position is a valid and not visited yet
                if   (   isValidMove(nextX,    nextY)    &&    visited[nextX][nextY]    ==
Integer.MIN_VALUE) {

                        visited[nextX][nextY] = moveCount;

                        if ( solveTraversal(moveCount + 1, nextX, nextY) ) {
                                System.out.println("Visited: ("+nextX+","+nextY+")");
                                return true;
                        }

 // BACKTRACK !!!
                        visited[nextX][nextY] = Integer.MIN_VALUE;
                }
        }
 return false;
}
```

```
        public booleanisValidMove(int x, int y) {
                if (x < 0 || x >= BOARD_SIZE || y < 0 || y >= BOARD_SIZE) {
                        return false;
                } else {
                        return true;
                }
        }

        public void printSolution() {
                System.out.println("The positions were visited as follows: ");
                for (int i = 0; i< BOARD_SIZE; i++) {
                        for (int j = 0; j < BOARD_SIZE; j++) {
                                System.out.print(visited[i][j] + " ");
                        }
                        System.out.println();
                }
        }
  }
```

Implementation.java

```
package solution;
import java.util.Scanner;

public class implementation {

        public static void main(String[] args) {
                Scanner sc= new Scanner(System.in);
                System.out.println("Enter the size of the chess board: ");
                int n=sc.nextInt();
                final int  chess_board_size = n;
                KnightTourknightTour = new KnightTour(chess_board_size);

                knightTour.solveKnightTourProblem();

        }

}
```