Jien Wei, Lim

Registration number 100276935

2023

The Knight's Tour Report

Supervised by Dr. Katharina Huber

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

The Knight's Tour problem is a classic puzzle where the goal is to construct a series of legal moves made by a knight so that it visits every square of a chessboard exactly once. This report presents the design and development of 2 applications that solves the Knight's Tour problem using either backtracking or Warnsdorff's algorithm. The report includes the discussion of the implementation of a GUI application, the outcome of the development, and a simulation study of the algorithms with and without the graphical user interface.

Acknowledgements

I would like to thank Dr Katharina Huber for providing continuous support and feedback in this project as well as providing clear and concise answers to any questions I had.

Contents

1 Introduction

The Knight's Tour is a mathematical problem where the objective is to find a sequence of moves made by a knight chess piece that visits every square of a chessboard exactly once. There are 2 variants of a knight's tour that can be created, which are the open tour and closed tour. A tour is considered closed if the final square that the knight lands on is one move away from the starting square (Figure 36), and open (Figure 37) if otherwise. This problem can be associated with graph theory as it is an instance of the more general Hamiltonian Path problem where finding an open or closed knight's tour is similar to finding a Hamiltonian Path or Hamiltonian Cycle respectively (Garey and Johnson, 1979).

The aim of this project is to create an application that is able to solve the Knight's Tour on an $r \times c$ chessboard, where r and c is the number of rows and columns on a chessboard respectively. 2 applications are created in this project: A game-like application that has a Graphical User Interface (GUI) to allow the user to interact with the program and displays the knight's step-by-step touring process, and a console line application suggested by Dr Katharina to be used for simulation which can find multiple tours and save them into files. The game application will be the main topic in the report as the simulation application is very similar to the game with the differences being having no GUI and capability of saving tours.

1.1 MoSCoW Requirements

To ensure that this project is completed successfully and can provide a satisfactory end-user experience, a prioritised requirements list was created. This ensures the fundamental elements that are key to the program's functionality are delivered before moving on to any other features that may be desirable.

Must:

- Find Knight's Tour using recursive Backtracking and Warnsdorff's algorithm.
- Chessboard can have variable row and column dimensions.
- Game application with a GUI that displays a chessboard, a knight chess piece, buttons for user input, and numbered steps to show tour.
- Game application can display each step a knight makes when finding tour.

- Simulation application allows user to provide keyboard input to create tours.
- Simulation application can save completed tours to a file.

Should:

- Backtracking algorithm is implemented as an iterative function.
- Both Game and Simulation applications are able to identify the type of tour.
- Game application displays lines between numbered steps of knight's tour to give ease of reading the Knight's Tour.
- Simulation application can save different types of tours to different files.

Could:

- Find Knight's Tour using a Divide and Conquer algorithm.
- Game application can change colour of numbered steps and lines.
- Game application can save and load incomplete tours.
- Game application can save completed tours to a file.

Won't:

- Find Knight's Tour using Euler's solution.
- Find Knight's Tour using multiple knights.

2 Background

The earliest surviving Knight's Tours came from (Murray, 1913), where the book describes an Arabic manuscript containing 2 tours on an 8×8 board, one given by al-Adir ar-Rumi, who flourished around 840 in Baghdad and is known to have written a book on an early form of chess played in the Middle East called Shatranj, and the other given by Ali C. Mani, an unknown chess player.

The first comprehensive mathematical analysis of the Knight's Tour problem was presented by Leonhard Euler (Euler, 1766), who proposed a solution of solving partially complete tours as well as constructing a closed tour from a completed open tour.

Both (Sandifer, 2006) and (Ball, 1917) provided an in-depth explanation regarding the step-by-step process of the solution. To summarize, the solution involves disconnecting paths between squares, and reconnecting them to create a new path to the unreachable squares. While the technique is feasible on paper, having to implement it is another story. Firstly, the solution assumes that the incomplete tours will have as few unreachable squares as possible. Due to the way the reconnecting is done, the solution would be computationally inefficient. The greater the number of unreachable squares, the longer it takes to complete the tour. Secondly, it is possible for the resulting tour to not start from the same square as the one used at the start. This technique applies to both the completion of incomplete tours and creating closed tours from opened ones.

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

(a) Before

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

(b) After

Figure 1: Incomplete tour finished using Euler's solution

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

(a) Before

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

(b) After

Figure 2: New tour with different starting position after using Euler's solution

In 1823, Warnsdorff (von Warnsdorf, 1823) proposed a greedy heuristic to the problem where the rule states:

> "Select the move which moves the knight to the square with the fewest number of further moves, providing that this number is not 0."

For example, if a knight is on position $(1,2)$ of the board, shown in Figure 3 coloured as a blue square, it has 3 squares (coloured green) that it can traversed to. On each square, they will have their own set of "future" squares that the knight can traverse to, as shown in Figure 4. By following Warnsdorff's rule, the knight will traverse to the blue square in 4a as it has the fewest number of future squares.

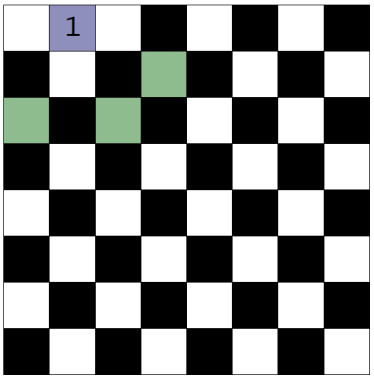
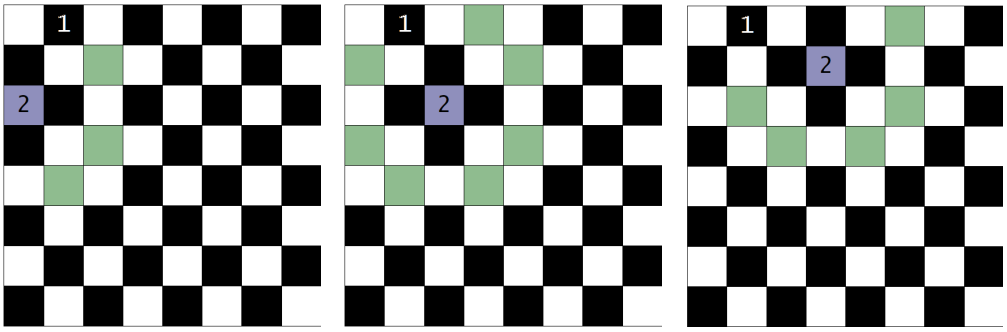


Figure 3: Starting square and its valid squares



(a) Square 1 (b) Square 2 (c) Square 3

Figure 4: Each valid squares' future squares

(Ghosh and Bhaduri, 2017) discussed solving the Knight's Tour using a recursive backtracking algorithm. Backtracking (Van Beek, 2006) is a depth-first search algorithmic method that tries every possible combinations of a solution until it finds one that can

solve the problem. For a chessboard sized $r \times c$, the time complexity of the algorithm in the worst case scenario would be exponential, or to be precise, $O(8^{r \times c})$. This is because a board has a total of $r \times c$ squares and a knight has a maximum of 8 possible moves to use. Due to it's extremely long run time, the algorithm is considered a brute force method for finding a Knight's Tour.

An interesting method to solve a Knight's Tour was proposed by (Conrad et al., 1994), where an $n \times n$ board can be solved using an algorithm with a linear time complexity as long as $n \geq 5$. This involves dividing the board into smaller square-shaped boards and r-shaped boards. In each of these boards, a knight's tour is generated and the tours are connected with the other tours on adjacent boards using the start and end points of the tours.

A Divide and Conquer algorithm proposed by (Parberry, 1997), is similar to (Conrad et al., 1994). If there is a large board, the board is divided into 4 smaller boards, where a premade knight's tour is generated in each board, then 4 edges are removed and added 4 edges are added to the inside corners to connect the boards. The algorithm is able to find a closed knight's tour on $n \times n$ or $n \times (n+2)$ boards in linear time if $n \geq 10$ and is an even number, as well as a closed knight's tour with one corner square not part of the tour in linear time if $n > 4$ and is an odd number. Whether this algorithm will be implemented or not will depend on the time remaining. Also, this algorithm relies on using a specific type of tour that has already been created beforehand in order to create the tour. This gave the idea of adding a save feature for different kinds of tours into separate files which can be used in the future for algorithms that connects tours together. The Divide and Conquer algorithm can only solve boards that have size $n \times n$, $n \times (n+1)$ and $n \times (n+2)$. It is unable to handle boards with an arbitrary size of $r \times c$. A linear time algorithm was proposed by Lin and Wei (2005) to solve that issue.

3 Methodology

3.1 Knight

The knight chess piece has a total of 8 L-shaped moves that can be made, which can be defined as:

$$(h,v) \in \{(\pm 2, \pm 1), (\pm 1, \pm 2), (\pm 2, \mp 1), (\pm 1, \mp 2)\}$$

where h and v are the number of squares the knight can move horizontally and vertically respectively. A negative or positive h value means that the knight will move to the left or right of the board respectively, while a negative or positive v value means that the knight will move upwards or downwards of the board respectively.

3.2 Chessboard

In chess, the standard size of a chessboard is 8×8 and each square on the board alternates between 2 colours, typically a light colour and a dark colour. 8×8 will be the default size of the board for the Knight's Tour, however, the dimensions can be changed to create smaller, larger, or rectangular boards. In this case, for a chessboard of size $r \times c$, $3 \leq r \leq 20$ and $4 \leq c \leq 20$, or $4 \leq r \leq 20$ and $3 \leq c \leq 20$. Also to note, the colour of the squares will not matter in finding a knight's tour and is purely for aesthetic purposes.

For every square on a board in a Knight's Tour, it can have 1 of 2 states: a Traversed state or Untraversed state. These states indicate whether the knight has already landed on the square or not. Secondly, for squares that have been traversed by the knight, it will have a number associated with it that represents the order of the sequence of squares that the knight has traversed. For example, if the top left corner square of the board is the 37^{th} square that the knight had traversed in the Knight's Tour, the square would have a value of 37.

3.3 Backtracking Algorithm

Instead of using recursion, I decided to implement the Backtrack algorithm as an iterative function. Although this increases the code complexity of the algorithm, it does avoid a computation overhead caused by recursion and allows for easier integration between the front end and back end of the game. This method requires a data structure that will act as the "memory" for the tour generation.

Every time the knight moves to a new square, the square is added to the memory. When the knight lands on a square that leads to a "roadblock", meaning that there are no valid moves to be made and there are still squares left to be traversed, it will go back to the previous square to find another valid move to use and traverse to a new square. And if that previous square has no valid moves, the knight will go back to the

square before it. The algorithm will continuously run to search for a knight's tour until a condition is met. If the knight lands on the final untraversed square on the board, the algorithm stops and a knight tour has been found successfully. However, if the knight has exhausted every possible moves for every square and sequence on the board and is still unable to traverse every untraversed board, it means that the algorithm has failed to find a knight's tour and will stop running. The pseudocode in Algorithm 1 shows the overall process.

Algorithm 1 Iterative Backtracking

Require:
Array M as Knight's Previous Moves
Array K as Knight's Possible Moves
2D Array B as Chessboard
while B contains untraversed squares **do**
  $(x, y, k) \leftarrow$ Last element in M
  for $i = k, \ldots, 8$ **do**
    $X_{New} \leftarrow x + K_i x$
    $Y_{New} \leftarrow y + K_i y$
    if (X_{New}, Y_{New}) is a valid and untraversed square **then**
      Last element of M $(x, y, k) \leftarrow (x, y, i+1)$
      Move Knight to $B_{(X_{New}, Y_{New})}$
      Add $(X_{New}, Y_{New}, 1)$ to M
      end for loop early
    end if
  end for
  if No valid moves found from (x, y) **then**
    Remove last traversed step from B
    Remove last element of M
    if No elements in M **then**
      Tour failed
    end if
  end if
end while
Tour found

3.4 Warnsdorff's Algorithm

Just like the Iterative Backtracking algorithm, Warnsdorff's algorithm will be implemented as an iterative function. However, instead of using a "memory", the algorithm will just restart the tour generation whenever it hits a roadblock. One interesting factor about Warnsdorff's algorithm is that different tours can be created from the same starting square. At times, it is possible to for the knight to land on a square where it will have more than 1 valid square with the same number of fewest future squares. When this situation happens, one of these squares will be selected at random. Warnsdorff's algorithm will also have 2 conditions which will stop it. If the knight lands on the final untraversed square of the board, it means that a knight's tour is found and the algorithm stops running. Since Warnsdorff's algorithm will restart whenever it hits a roadblock rather than going back to a previous square, the condition to stop running for a failed knight's tour generation is changed where if the algorithm restarts too many times, it would indicate a knight's tour is unable to be found and the algorithm stops. The pseudocode in Algorithm 2 shows the general process of the algorithm.

3.5 Game GUI

For the overall layout of the GUI, I chose a simple design for it by using as few images as possible and relying only on shapes to create every component of the GUI, including the board. The size and position of the components are affected by the screen resolution of the machine they are running in. The bigger the screen resolution, the bigger the buttons will look. Figure 5 shows the initial design of the GUI.

The rectangles in the image indicate buttons that can be clicked on by the user to interact with the game. As the program requires the user to click on a square of the chessboard to find a knight's tour, the squares themselves can be considered as buttons. For the dotted-lined rectangle, it means that the button is non-visible to the user. This was the initial plan of the implementation of the algorithm selection method. For example, if the user clicks on the button labeled "Backtrack Method", the button will not be visible and the button labeled "Warnsdorff's Method" will appear and can be clicked on.

Following the advice given by Dr Katharina, the GUI's algorithm selection method was changed to make it more user friendly. Instead of hiding one of the buttons to show which algorithm can be selected, a much larger button labeled Algorithms is displayed

Algorithm 2 Warnsdorff's Algorithm

Require:

Array K as Knight's Possible Moves
2D Array B as Chessboard
(P_x, P_y) as Knight's Current Position
while B contains untraversed squares **do**
  for Every move, R, from K at random **do**
    $X_{New} \leftarrow P_x + R_x$
    $Y_{New} \leftarrow P_y + R_y$
    if (X_{New}, Y_{New}) is a valid and untraversed square **then**
      if (X_{New}, Y_{New}) has fewest untraversed, connected squares **then**
        $(P_x, P_y) \leftarrow (X_{New}, Y_{New})$
        Move Knight to $B_{(X_{New}, Y_{New})}$
      end if
    end if
  end for
  if No valid moves found from (P_x, P_y) **then**
    if Tour restarted too many times **then**
      Tour failed
    else
      Restart tour
    end if
  end if
end while
Tour found

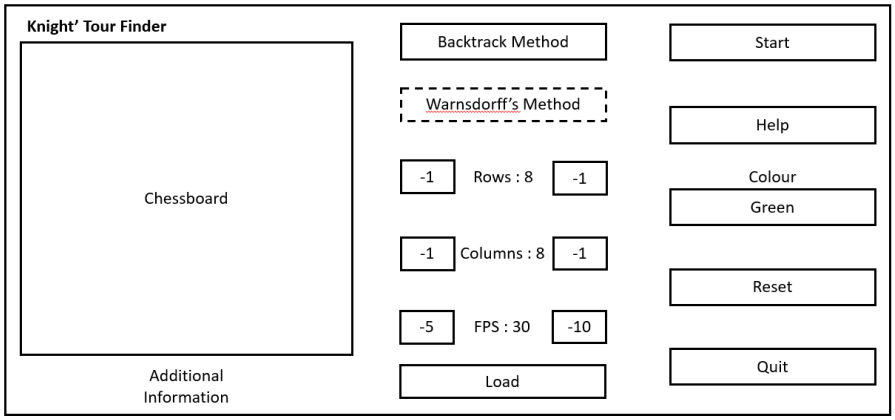


Figure 5: Planned design of the game's GUI

on the screen. When the user clicks on Algorithms, it will be removed and shows both the Backtrack and Warnsdorff buttons and the user can click on either one to choose the algorithm. Once clicked, the Algorithm button will reappear with text showing the current algorithm.

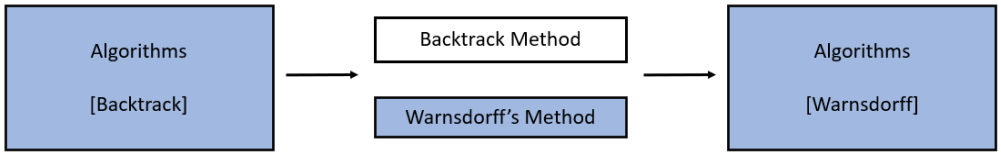


Figure 6: Selecting Warnsdorff's Algorithm

Although using a simple image of an 8×8 chessboard would be enough to display the chessboard on the GUI, it will not work for this program. As the board can have different number of rows and/or columns, the GUI requires a way create a board with sizes other than 8×8. For that, the only way that it can be feasible was by drawing each and every square to create the board.

3.6 Giving Instructions

In the game, it is possible that the user will not know how to navigate their way in it. Hence, a Help section is needed to be created to give information to the user regarding the features in the game and the step-by-step process of finding the knight's tour. The GUI will display a Help button where the user can click on to display a pop up that will

show the necessary information.

3.7 Frames Per Second (FPS)

In the game application, the user will be allowed to change the frame rate of the board so that the speed of the tour generation can be changed. The frame rate should only affect the board and not the entire game as whole and can be changed at any time during the program, except after a tour has been found. The minimum FPS that the tour can run on is 1 while the maximum value is 60. However, it is possible to set the FPS so that it is determined by the computer's processing power.

3.8 Simulation

The console line application, which will be called a simulation application from now on, will allow the user choose either Backtrack or Warnsdorff algorithm to find the Knight's Tour on chessboard with dimensions specified by the user. Unlike the game where the knight generation process can be shown, the simulation will only show the final result of the tour and create a number of successful tours, which is specified by the user, before stopping. Not only that, the simulation will be able to save the successful tours into files. Rather than just saving successful tours into all into one file, the simulation will be capable of saving different kinds of tours and/or other data into separate files.

Data that can be saved and file format to be used:

- Successful tours (Text file)
- Time and steps taken to generate tour (CSV file)
- Open or Closed tours in separate files (Text file)
- Structured or Unstructured tours in separate files (Text file)

Before starting the tour generation, the simulation will need to receive some keyboard inputs from the user.

List of inputs required by the user:

1. Number of rows for the board. (Minimum 3, Maximum 20)
2. Number of columns for the board. (Minimum 3, Maximum 20)

3. Type of algorithm to find knight's tour. (Backtrack or Warnsdorff)
4. Type of simulation. (all, random, specific)
5. Starting square on the board
6. Number of successful tours to find before stopping.
7. Choice of saving successful tours to a file.
8. Choice of saving time taken to find a tour to a file.
9. Choice of saving open or closed tours to separate files.
10. Choice of saving structured or unstructured tours to separate files.

Type of simulation and what they do:

- all - Use every square of the board once as the starting position to find a knight's tour.
- random - Choose a random square of the board as the starting position to find a knight's tour.
- specific - Use a square specified by the user as the starting position to find a knight's tour.

Once the user provides all of the relevant information, the simulation will start finding a tour and print out successful tours on the console line and/or save them to the appropriate files. If the Backtrack algorithm fails to find a tour once, it will stop the touring process as no new tours can be created even after resetting. As for the Warnsdorff algorithm, failing to find a tour will reset the tour generation process and add a point to a counter. When the counter reaches a certain threshold, the touring process will stop.

4 Implementation

4.1 Prep Work

Due to its data structure handling and extensive libraries (Srinath, 2017), Python was chosen to be used as the backbone of the program and Pygame, Python's graphics library

for video games (McGugan, 2007), was used to handle the GUI. The next task was to import the prerequisite libraries that will be used globally by the game. The list of libraries imported are listed below.

sys Only used to stop the application if user wishes to exit the game.

pygame Python's library to create the game's GUI.

numpy Python's array library.

random Pseudo-random number generator.

json Library that handles JSON objects and files.

ctypes Used for properly fitting the GUI onto the display.

datetime Used for recording the time taken to find a Knight's Tour.

The *pygame* and *numpy* libraries are given aliases *pg* and *np* to slightly reduce the length of every line of code that uses them. For reference, the Pygame library was used at least 100 times in order to call its functions in the game application's code. The next step was creating classes separating the chessboard, knight, and game. The classes are named Board, Knight, and GameState respectively.

4.2 Knight

Referring back to 3.1, a knight has a total of 8 legal moves that it can make on a square. Since the knight cannot add new moves or lose existing ones, tuples will be used to create and store all the legal moves. Tuples are similar to arrays with the notable differences being tuples are not capable to add, remove, or change elements inside it, as well as being an ordered list meaning elements are unable to be shuffled. When the knight wishes to traverse to a new square in the Backtrack algorithm, it will have to read through the tuple in the same order every time and get the first valid move it finds. The main use of the tuple is to maintain consistency for the Backtracking algorithm when finding a knight's tour. Figure 7 shows the creation of the Knight class with its list of legal moves as well as other attributes associated with it for the knight's tour.

```
class Knight:
    def __init__(self):
        self.knight_moves = ((2, 1), (1, 2), (-1, 2), (-2, 1), (-2, -1), (-1, -2), (1, -2), (2, -1))
        self.knight_initial_pos = None  # The first square that the knight starts on.
        self.knight_pos = None  # The current square that the knight is on.
        self.knight_step = 1  # The furthest step of the knight's tour the knight is on.
        self.move_log = []  # An array storing the squares that the knight has traversed
        self.total_steps = 0  # Total number of times a Knight has moved
```

Figure 7: The Knight class

4.3 Board

In Python, the chessboard is represented as a 2-Dimensional array or an array of arrays where each array is a row on the board and each cell of the array represents a square on the board. Rather than using Python's standard arrays, Python's Numpy library was used instead as it contains functions that will make the creation of the board array easier. Going back to 3.2, every square in a chessboard has 2 key pieces of information, state and tour order number. Given as advice by Dr Katharina, instead of using 2 arrays to contain them separately, it is possible to use 1 array to represent the information and the board as a whole.

Firstly, a square can only have 2 states, traversed (T) and non-traversed (NT). Putting that in a programming stand point, it would mean that whether a cell in an array has been traversed can either be True or False. In Python, the True and False boolean values are also numerical values 1 and 0. This means a non-traversed board will contain only 0s while a fully-traversed board will contain only 1s. Secondly, the order number of each square can range from 1 to the total number of squares in the board. These numbers are all non-zero values, which means that as long as the number in a cell is greater than 0, it will indicate that the cell/square has been traversed by the knight.

To create the chessboard array, Numpy's *zeros()* function will be used which will create a 2-D array of size $r \times c$ filled with 0s. This board array will be labeled as *graph*. Another 2-D array, labeled as *board_moves*, is created using the same function. This is to store the total number of times a knight has traversed each square, which will be displayed for the user to see. The code to create both arrays is:

np.zeros([r, c], dtype=int)

where r is the number of rows for the board, c is the number of columns for the baord, and *dtype=int* specifies that the array will only contain integer values.

```
[[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]]
```

Figure 8: A newly created 8×8 board array

For *move_log*, it is an array that stores the previous squares that the knight has traversed. Each element holds 3 pieces of information, the row and column number of the square, and the index of the *knight_moves* array. The *move_log* array acts as the memory in the Backtracking algorithm which will be explained in 4.8.

4.4 GameState (Game GUI)

In order to start up Pygame, the *init()* function is needed to be called. This will only initialise the library and does not create the game window application. To do that, the *set_mode()* function in Pygame's *display* module is needed to be called. By passing Pygame's constant variable, FULLSCREEN, into *set_mode()*, the game window will be in full screen mode. As the components of the GUI will be affected by screen resolution, the width and height of the display will need to be obtained by using the *get_size()* function. To change the colour of the game window's background, the *fill()* function can be used.

Although these few lines of code should be enough to display the game windows, there was a slight issue relating to Windows machines where the screen resolution values were not obtained properly and the GUI ended up not fitting onto the screen. This only

```
class Board:
    # The default size of a chessboard is 8 x 8
    def __init__(self, row_dimension=8, col_dimension=8):
        self.knight = Knight()  # Every board requires a Knight for a knight's tour
        self.row_dimension = row_dimension  # Number of rows in the board
        self.col_dimension = col_dimension  # Number of columns in the board
        # 2D array representation of board
        self.graph = np.zeros([row_dimension, col_dimension], dtype=int)
        # 2D array to store number of times Knight traversed each square
        self.board_moves = np.zeros([row_dimension, col_dimension], dtype=int)
```

Figure 9: The Board class

```
BACKGROUND_COLOUR = (180, 241, 255)
BUTTON_COLOUR = (100, 100, 100)  # Default button colour
HOVER_BUTTON_COLOUR = (170, 170, 170)  # Colour of button when cursor hovers over
BUTTON_TEXT_COLOUR = (255, 255, 255)  # Colour of text inside buttons
TEXT_COLOUR = (0, 0, 0)  # Colour of text not in buttons
# Colours for move stamps and lines
STAMP_COLOURS = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (144, 142, 0)]  # Red, Green, Blue, Colour Blind
# Colours for text of move numbers inside stamps
MOVE_COLOURS = [(0, 0, 0), (0, 0, 0), (255, 255, 255), (255, 255, 255)]
```

Figure 10: Variables containing colour information for GUI components

happened when running the executable file but not for the Python code. Using the *ctypes* library, the issue can be fixed using one line of code shown in Figure 12, written before the *init()* function is called.

One thing to note about the way Pygame works is that when drawing objects onto the display, they are not immediately shown. Instead, Pygame's *update()* function in its *display* module is needed to be called to update the display and show it on the screen. This means that a continuous loop is needed to display any changes to the GUI. This loop can be created by using an infinite *while* loop that will be the game's main driver.

4.4.1 Rectangle Class

For all of the buttons and squares of the board, they are created by drawing rectangle shapes onto the display. To draw them, the Pygame's Rect class is first used to create the rectangle object (*Rect(left, top, width, height)*). *left* and *top* are the positions of the top left corner of the rectangle on the x and y axis of the display respectively, as Rect objects are always drawn using the top left corner of the shape as the starting point. *width* and *height* determines how long and tall the Rect object will be. After the Rect object is created, it can then be drawn onto the display using the *draw.rect(surface, colour, rect)*

```
# Start up the game windows
pg.init()
SCREEN = pg.display.set_mode((0, 0), pg.FULLSCREEN)  # Set game window
SCREEN.fill(BACKGROUND_COLOUR)  # Set background color
x_axis, y_axis = SCREEN.get_size()
```

Figure 11: Starting up game window

```
# Used to fit display into correct resolution
ctypes.windll.user32.SetProcessDPIAware()
```

Figure 12: Code to fix executable's GUI display issue in Windows

function where *surface* would be the *SCREEN* variable initialised previously, shown in Figure 12, *colour* is the colour of the rectangle, and *rect* is the Rect object.

To display text in the GUI, a *SysFont* object from Pygame's *font* has to be created which will then need to be rendered by calling the object's *render()* function. Lastly, the *get_rect()* function is called from the rendered object to position the text in the middle of the text area.

With so much information to use for one component and the GUI having so many objects to draw, a Rectangle class was created in Components.py to store all of the information into one object to make modifying and accessing the information easier.

4.4.2 Chessboard

As the chessboard is able to have its dimensions modified by the user, using an image of a standard 8×8 chessboard to display it on the GUI will not be feasible. Because of this, the only way to draw the board is by drawing each and every square of the board using the Rectangle class. The chessboard is connected to both the *graph* and *board_moves* arrays. The squares are drawn using 2 *for* loops where the squares are drawn column by row. This means that the order of the squares drawn goes from left to right first, then top to bottom. One very important thing to note when drawing the squares is the axis used to calculate the position of the squares. Figure 15 shows an example. The starting point of Square $(1, 1)$ is simple enough since it starts as the first square of the board. The width and height are also straight forward since they use the x and y axis respectively to determine the size of the square. For the other squares, however, their positions can

```python
class Rectangle:
    def __init__(self, x_pos, y_pos, width, height, pg, colour=(180, 241, 255), hover_colour=None,
                 text=None, text_colour=None, text_font=None):
        self.x_pos = x_pos  # x-axis position of the top left corner of rectangle
        self.y_pos = y_pos  # y-axis position of the top left corner of rectangle
        self.width = width  # Width of rectangle
        self.height = height  # Height of rectangle
        self.colour = colour  # Colour of rectangle
        self.hover_colour = hover_colour  # Colour of rectangle when mouse hovers over it
        self.text_colour = text_colour  # Colour of text inside rectangle
        self.rect = pg.Rect(x_pos, y_pos, width, height)  # Create Pygame Rect object
        self.text = text  # Text string inside of rectangle
        self.text_font = None  # Font of text. Uses Pygame's SysFont function
        self.text_render = None  # Rendered text object
        self.text_rect = None  # Text Rect object
        # Initialise the Text Rect object if there contains text to be displayed
        if text is not None:
            self.text_font = text_font
            self.text_render = text_font.render(text, True, text_colour)
            # Centers the text in the middle of the Rectangle
            self.text_rect = self.text_render.get_rect(center=(x_pos + (width // 2), y_pos + (height // 2)))
```

Figure 13: The Rectangle class

```python
def change_text(self, text):
    self.text = text
    self.text_render = self.text_font.render(self.text, True, self.text_colour)
    self.text_rect = self.text_render.get_rect(
        center=(self.x_pos + (self.width // 2), self.y_pos + (self.height // 2))
    )
```

Figure 14: Function for updating text

be a little bit tricky to grasp at first. Looking at Square $(5,2)$, it is in the 5^{th} row and 2^{nd} column of the board. The greater the row number, the lower down the board the square is located. The same goes for column number. Currently, the square is in column 2, located closer to the left. The position of the square on the x and y axis is determined by the height and width of the square respectively.

$$Square_x = Square_{Row} \times Square_{Height}$$

$$Square_y = Square_{Column} \times Square_{Width}$$

The chessboard's coloring of the squares is done by calculating the sum of the row and column number of each cell in the *graph* array and checking whether it is even or odd. Even sum cells means that the squares are coloured white while odd sum ones are gray.

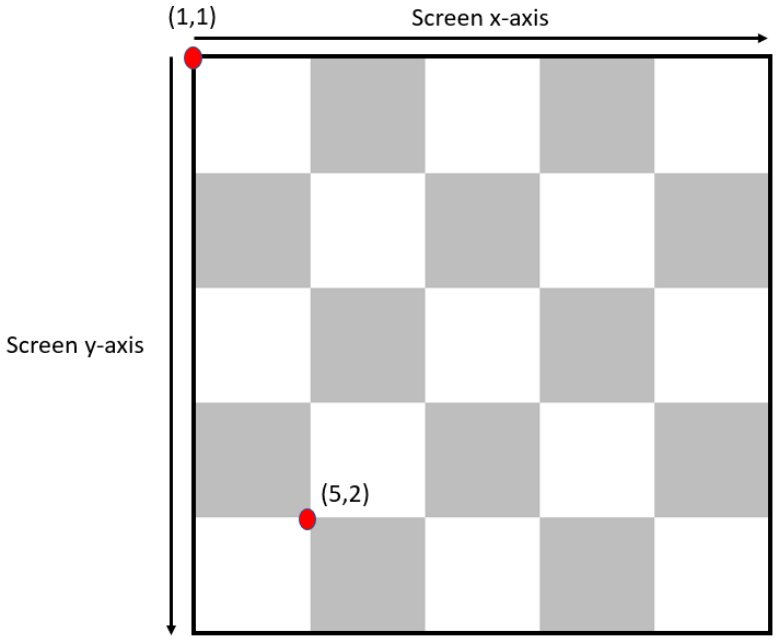
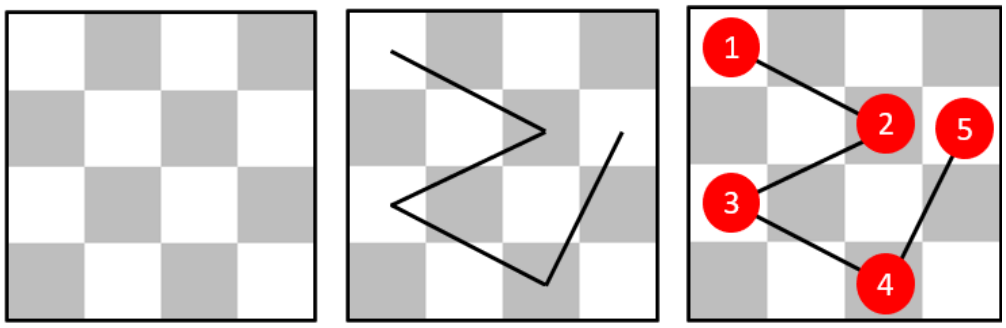


Figure 15: Drawing the board

When the knight moves around the chessboard, it will leave a trail of numbered stamps and lines indicating the sequence order of the knight's tour. These components can be separated into layers which will be drawn in a specific order to display the sequence. Whenever the board needs to be redrawn, a Rect object with the size of an 8×8 chessboard with the same colour as the background is first drawn over the current board. This is mainly used for boards smaller than 8×8 since drawn objects cannot erase themselves from the GUI. After "clearing" the area, the first layer, which are the squares of the board, are drawn to create the chessboard (Figure 16a). After that, the next layer of components, the lines, are drawn on top of the squares (Figure 16b). These lines are only drawn between traversed squares that are connected with each other. Lastly, are the numbered stamps, these stamps are circles which are drawn on traversed squares using the *circle()* function in Pygame's *draw* module. On top of the stamps, are the number texts indicating the order in which the knight traversed the squares (Figure 16c).

To draw the knight, the program will find the square containing the largest step number. Whichever square has the largest number, will have a knight drawn inside it unless the largest number equals to the total number of squares on the board, in which case a numbered stamp will be drawn in the square instead. To draw the knight, the knight

image is first loaded into the Pygame window using the *load()* function in the *image* module and depending on the size of the squares on the board, the image will be resized using the *scale()* function in Pygame's *transform* module to fit the image inside the squares. Finally, the *blit()* function is used to draw the image onto the display by passing the loaded image object and position of the image. The function to handle the redrawing of the board is defined as *redraw_board* and its pseudocode is shown in Algorithm 3.



(a) Board first (b) Lines second (c) Stamps third

Figure 16: Drawing the tour sequence

Algorithm 3 Redraw Board during Tour Generation

```
Step_Max ← Largest value in board
Draw chessboard
Draw lines
Draw numbered stamps
if Step_Max = Total number of squares on board then
    Draw final numbered stamp on square
else
    Draw knight image on square
end if
```

For chessboards of size $r \times c$ where $3 \leq r < 8$ and $3 \leq c < 8$, the GUI will merely remove any rows or columns from the board and redraw the board which will look smaller compared to the standard 8×8 board. When $8 \leq r \leq 20$ and $8 \leq c \leq 20$, the size of the chessboard will not increase. Instead, the size of the squares in the chessboard

will be scaled to fit inside the board. Inside the squares, the knight image, numbered stamps, and text will also be scaled to fit inside the modified squares.

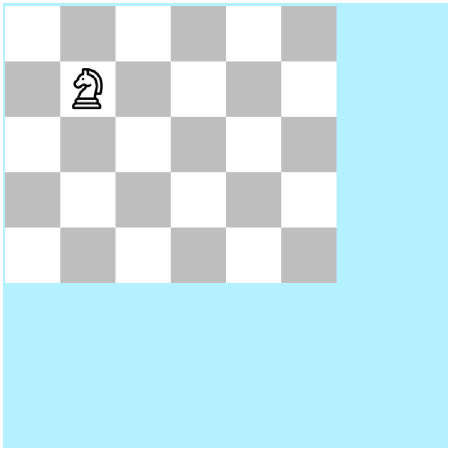


Figure 17: 5×6 table

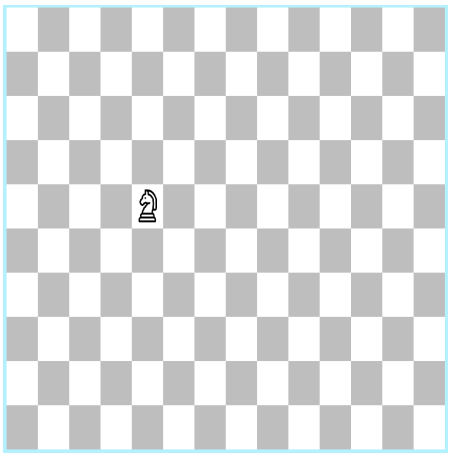


Figure 18: 10×14 table

4.4.3 Help

In order for the user to be able to understand what each button does and knowing how to create a knight's tour when using the game, a "tutorial" is needed to be created. In the game, there will be a button labeled "Help" that will be displayed in the GUI. When the user clicks on it, a pop up will appear giving a description of what the board and buttons

do. In the pop up, there will be 3 pages that the user can navigate through: The first is the description of the board and buttons, the second is the process to create a knight's tour, and the third is saving and loading an incomplete tour. In every page of the pop up, there will be the main "Help" title, a subtitle of what the given information is about, a "wall" of text providing the user information, the current page number the user is on, and buttons to traverse to different pages and exiting the Help pop up. Figure 19 shows an example of the help pop up.

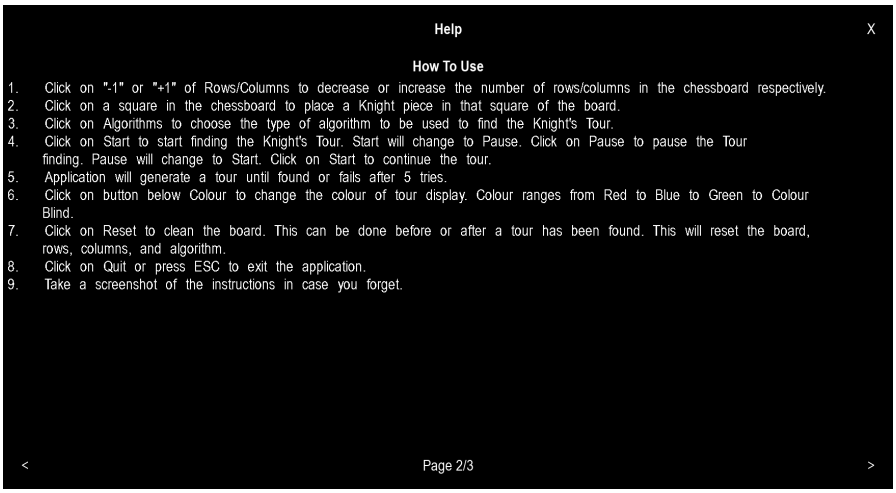


Figure 19: Page 2 of the Help Pop Up

Unfortunately, Pygame is not capable of creating a single text object containing different pieces of text with different styles. Because of that, every single component in the pop up had to be created separately using the Rectangle class. Figure 20 shows how the help pop up is created by using separate Rectangle objects. Both rectangles with red and yellow outlines are Rectangle objects which are created and connected together to give the illusion that it is just one big block. For the rectangles outlined with red, these are buttons that the user can click on. Rather than calculating their positions and dimensions to connect with the other components, they are layered over the components to reduce the complexity of aligning them in different screen resolutions. In this case, the exit button labeled, as "X", is layered over the Help title Rectangle object, while the previous-next page buttons, labeled as "<" and ">" respectively, are layered over the Page number Rectangle object.

To display the "wall" of text that is the information for the user, rather than writing the information inside the code, I decided to use text files which the program can read

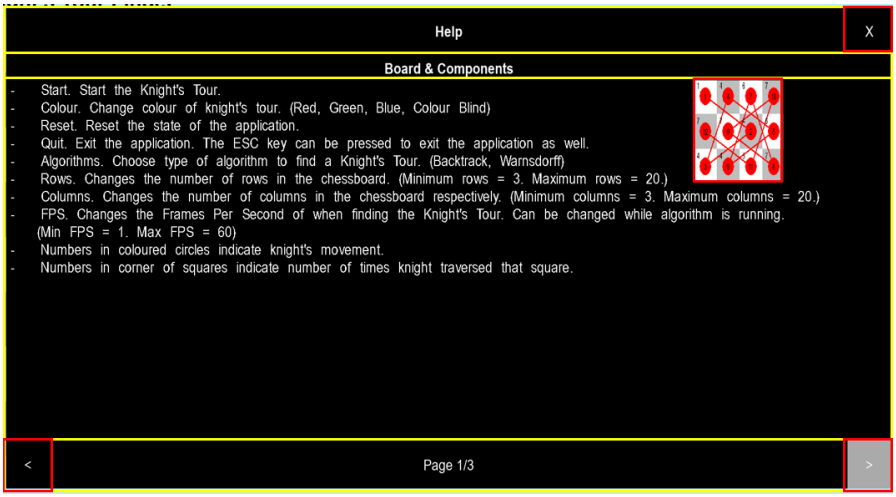


Figure 20: Everything's a Rectangle object!

from in order to display the the information. This would avoid having to recompile the code and recreating the executable file every time any changes to the text is made. Since there are 3 pages of Help information, 3 text files were created. When the game starts up, it will look for the 3 text files and extract the text from them to be stored in separate variables. When the user clicks on the Help button, a variable will be accessed to display the text on the pop up. Which variable that will be used to display the text is dependent on which page the user is currently on.

4.4.4 Frames Per Second (FPS)

The frame rate of the board affects the speed of the tour generation and the default frame rate will be 30 FPS when the game starts up. If the user wishes to decrease or increase the speed of the tour generation, they will need to decrease or increase the frame rate respectively. The minimum frame rate is 1 while the maximum is 60, however, if the user tries to increase the frame rate above 60, the game will forgo the frame rate limit set and use the maximum frame rate dependent on the computer's processing power. Between a frame rate of 1 to 10 FPS, the FPS will increase and decrease by intervals of 1. Between 10 to 30 FPS, the interval will be by a value of 5. Finally, between 30 to 60 FPS, the interval to increase and decrease the FPS will be by a value of 10.

To control the speed at which the tour generation is done, the *get_ticks()* function from Pygame's *time* module will be used. The function returns the number of milliseconds

Algorithm 4 Change Board FPS

Require: Integer FPS
  if $1 \leq FPS < 10$ **then**
    $FPS \leftarrow FPS + 1$
    OR
    $FPS \leftarrow FPS - 1$
  end if
  if $10 \leq FPS < 30$ **then**
    $FPS \leftarrow FPS + 5$
    OR
    $FPS \leftarrow FPS - 5$
  end if
  if $30 \leq FPS \leq 60$ **then**
    $FPS \leftarrow FPS + 10$
    OR
    $FPS \leftarrow FPS - 10$
  end if

(ticks) since Pygame was initialised. When the tour starts, the number of ticks is first obtained and stored in a variable. When the knight finishes one step of the touring process, it will perform 2 calculations. The first calculates the difference between the current and previous number of ticks. The second takes 1000 milliseconds and divides it by the FPS value, giving the total number of milliseconds required before the frame can be updated. The difference value of the ticks is then compared with the milliseconds required. If the difference is greater than the required value, it means that the time between each frame has passed and the display can be updated. Else, no operations will be done until the condition is satisfied, which also means the touring process will be halted temporarily until the board is updated.

4.5 Simulation

For the simulation application, rather than having the user change the code internally in order to perform the tour generation, the *input()* is used to receive keyboard inputs from the user to provide parameters for the application to run. The application will display a

Algorithm 5 Board Redraw Speed

Require: Integer FPS
    $FPS_{Max} \leftarrow 60$
    if $FPS > FPS_{Max}$ **then**
        Redraw board
    else
        $Ticks \leftarrow Ticks_{Current} - Ticks_{Previous}$
        $Milliseconds \leftarrow 1000/FPS$
        if $Ticks > Milliseconds$ **then**
            Redraw board
            $Ticks_{Previous} \leftarrow$ Current time in milliseconds
        end if
    end if

prompt asking for input and the type of input the user can provide. On the off chance of user error and an invalid input is given, the application will show an error message and prompt the user to try again. For example, if the user is asked to give a positive integer value between 3 and 20 for the chessboard's row dimension and they input a decimal number, negative number, number not in range, or a string, the application will perform error handling and asks the user to try again with the same prompt.

Going back to the list of inputs in 3.8, some information are not needed to be given by the user depending on the type of simulation they chose.

Table 1: Information required by simulation types

	all	random	specific
Starting Square position	No	No	Yes
Number of tours to generate	No	Yes	Yes

4.6 Checking For Valid Squares

When the knight moves around the board, a function is used to check whether the square the knight is moving to is valid. The function takes the knight's current position and a

legal move from the list, and calculates the position of the next square. If the position of the square is out of range of the dimensions of the array, it means that the square is not valid. If the square is valid, but the value it holds is non-zero, it means that the square has already been traversed and not valid to move to.

```
def is_valid_move(self, x, y):
    """
    A utility function to check if square (x, y) is valid and untraversed on the chessboard
    :param x: Row number of square
    :param y: Column number of square
    """
    if 0 <= x < self.row_dimension and 0 <= y < self.col_dimension and self.graph[x][y] == 0:
        return True
    return False
```

Figure 21: Function for checking validity of squares implemented in Python

4.7 Counting Empty Squares

For Warnsdorff's algorithm, not only must squares be checked for validity in order for the knight to move, the number of future squares must be checked on squares that can be traversed so that the knight only moves to the square with the least number of future squares. This can be done by reusing the *is_valid_move()* function shown in Figure 21. From the current square,

```
def count_empty_squares(self, next_x, next_y):
    """
    Counts number of valid and untraversed squares from square (next_x, next_y)
    :param next_x: Row number of square
    :param next_y: Column number of square
    :return:
    """
    count = 0
    for i in range(8):
        if self.is_valid_move(next_x + self.knight_moves[i][0], next_y + self.knight_moves[i][1]):
            count += 1
    return count
```

Figure 22: Function for counting future squares implemented in Python

4.8 Backtracking Algorithm

This section will be explaining how the function for Backtracking works on every iteration. The primary loop is used to check whether the knight's step count reaches the total number of squares on the board. If reached, it means a tour has been found and the loop stops. The function does not require any parameters passed through as it can access the variables contained in the Knight, Board, and GameState classes. The main variable the function uses is the *move_log* array where the elements contain 3 pieces of data, (x, y, i), where x and y are the row and column number of a traversed square and i is the index number for the *knight_moves* array.

The function first takes the last element of the *move_log* array and gets the reads the i value inside of the element. Starting from the i^{th} index of *knight_moves* array, the elements in *knight_moves* are used to check whether they move the knight to a valid square by using the function defined in Section 4.6. If a valid move is found, the knight adds one to its number of steps and the last element in *move_log* is updated so that $i+1$. This is so that when the element is accessed again, it will start at index $i+1$ of *knight_moves* rather than i. This is to prevent a deadlock if the algorithm goes back to the element to search for another valid move. After that, the knight uses the valid move and its current position to calculate the new square position for it to traverse to. The knight then moves to the new square and adds its step number to the square. A new element consisting of the new square's position and 0 is added to *move_log*. When the next iteration happens, this new element would be the last element that will be accessed by the function and the *knight_moves* array will be read starting from index 0.

If none of the elements in *knight_moves* allow the knight to make a valid move, it would have to go back by one square to search for another move to use. To do that, the function first sets the square the knight is currently on to a value of 0, reverting it back as an untraversed square and the knight will lose a step. The last element is then removed from *move_log*. If there are still elements in *move_log*, the knight's position is updated to the new last element's square position. However, if the last element is removed and *move_log* ends up being empty, it means that all possible moves on every square in every sequence have been used and a knight's tour is still unable to be found. If that happens, the function will completely stop finding a knight's tour. Figure 23 shows the code written in Python to implement the Backtracking algorithm.

```
def find_tour_backtrack_iterative(self):
    while self.knight_step < self.row_dimension * self.col_dimension:
        last_used_square = self.move_log[-1]
        contains_valid = False
        self.total_steps += 1
        for i in range(last_used_square[2], 8):
            new_x = last_used_square[0] + self.knight_moves[i][0]
            new_y = last_used_square[1] + self.knight_moves[i][1]
            if self.is_valid_move(new_x, new_y):
                self.move_log[-1] = (self.knight_pos[0], self.knight_pos[1], i + 1)
                self.knight_step += 1
                self.graph[new_x][new_y] = self.knight_step
                self.moves[new_x][new_y] += 1
                self.knight_pos = (new_x, new_y)
                new_pos = (new_x, new_y, 0)
                self.move_log.append(new_pos)
                contains_valid = True
                break
        if not contains_valid:
            self.graph[self.knight_pos[0]][self.knight_pos[1]] = 0
            self.knight_step -= 1
            self.move_log.pop()
            if len(self.move_log) == 0:
                print(len(self.move_log))
                self.tour_found = False
                self.tour_failures += 1
                return False
            self.knight_pos = (self.move_log[-1][0], self.move_log[-1][1])
    self.tour_found = True
    return True
```

Figure 23: Backtracking Algorithm implemented in Python

4.9 Warnsdorff's Algorithm

Just like in Section 4.8, this section explains the function for the Warnsdorff's algorithm on every iteration, and the primary loop functions just like the Backtracking algorithm. The function also does not require any parameters and rely on variables held by the Knight, Board, and GameState objects. The function first starts by initialising 2 variables: *least_empty* containing 9 and *least_empty_index* containing -1. *least_empty_index* holds the index number of the move located in *knight_moves* which leads to the fewest future squares while *least_empty* holds the number of future squares. Since the maximum number of future squares a square can have is 8, *least_empty* is set

to 9 so that the 2 variables will always get the first move's index and future squares to compare with other moves.

Using the *randint()* function, a random integer between 0 and 8 is obtained which will be the index number of the element in *knight_moves* to be read first. Starting from that element, the *knight_moves* array is looped through until reaching the final element, which will be located before the starting element used. For every element in *knight_moves*, the new square position is calculated and the number of future squares from that new square is counted using the function defined in Section 4.7. If the number of future squares counted is lower than *least_empty*, *least_empty* is updated to hold the value of the counted future squares and *least_empty_index* is updated hold the index number of the current element in *knight_moves*. This repeats until the last element is reached. If *least_empty* remains to be -1, it means that the knight has hit a roadblock, unable to move further, and failed to find a tour. If that happens the function will reset the board to find a tour again until it fails to find a tour 5 times.

If *least_empty* has been modified even once, it means there is a valid move. The position of the new square is calculated using the current position of the knight and element of *knight_moves* at index number stored in *least_empty_index*. The knight will add a step to itself and move to the new square. This repeats until the board is fully traversed or the knight hits a roadblock.

4.10 Calculating Time

For both game and simulation, when a knight's tour is successfully found, the time taken to find it is calculated which will either be displayed on the screen or saved into a file. To calculate the duration, the start time of when the touring generation starts is first recorded and stored in a variable using the *now()* function from the *datetime* library. Once the touring generation finishes, *now()* is called again to get the current time and the difference between the current time and start time is calculated to get the total number of seconds. This will be the duration for the touring process. The same principle applies when the tour generation is paused. When paused, the duration is calculated and added to the total. When resumed, the current time becomes the start time.

Algorithm 6 Calculate Tour Generation Time

$Duration \leftarrow 0$
while Program runs **do**
  if New tour starts **then**
    $Time_{Start} \leftarrow$ Current time
  end if
  if Tour generation is paused **then**
    $Time_{Current} \leftarrow$ Current time
    $Duration \leftarrow Time_{Current} - Time_{Start}$
  end if
  if Paused tour is resumed **then**
    $Time_{Start} \leftarrow$ Current time
  end if
  if Tour generation ends **then**
    $Time_{Current} \leftarrow$ Current time
    $Duration \leftarrow Time_{Current} - Time_{Start}$
  end if
  if Game or Tour is Reset **then**
    $Duration \leftarrow 0$
  end if
end while

```
def find_tour_warnsdorff(self):
    while self.knight_step < self.row_dimension * self.col_dimension:
        least_empty = 9
        least_empty_index = -1
        random_num = randint(0, 1000) % 8
        for i in range(8):
            index = (random_num + i) % 8
            new_x = self.knight_pos[0] + self.knight_moves[index][0]
            new_y = self.knight_pos[1] + self.knight_moves[index][1]
            empty_sq_count = self.count_empty_squares(new_x, new_y)
            if self.is_valid_move(new_x, new_y) and empty_sq_count < least_empty:
                least_empty_index = index
                least_empty = empty_sq_count
        if least_empty_index == -1:
            self.tour_found = False
            self.tour_failures += 1
            return False

        new_x = self.knight_pos[0] + self.knight_moves[least_empty_index][0]
        new_y = self.knight_pos[1] + self.knight_moves[least_empty_index][1]
        self.knight_step += 1
        self.total_steps += 1
        self.graph[new_x][new_y] = self.knight_step
        self.moves[new_x][new_y] += 1
        self.knight_pos = (new_x, new_y)

    self.tour_found = True
    return True
```

Figure 24: Warnsdorff's Algorithm implemented in Python

4.11 Detecting Types of Tours

For both game and simulation, a function was implemented which will check whether the tour generated was opened or closed. After a knight's tour is generated, the function will take the positions of the first and last squares of the tour and calculate the number of squares between them.

$$Squares_h = |Final_X - Start_X|$$

$$Squares_v = |Final_Y - Start_Y|$$

If $Squares_h = 2$ and $Squares_v = 1$ or $Squares_h = 1$ and $Squares_v = 2$, it means that the knight can move to the starting square from the final square of the tour, indicating

that it is a closed knight's tour.

In the simulation, it contains a function that checks whether a tour is structured or unstructured. A tour is considered structured if it contains a specific set of knight moves in each corner of the board as shown in Figure 25, and unstructured if not. This function is mainly to be used for the Divide and Conquer algorithm.

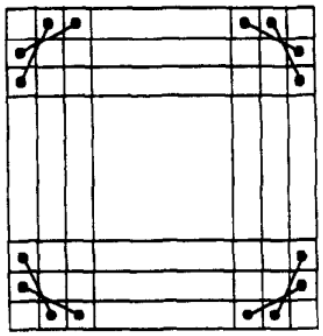


Figure 25: Structured tour's required moves

```python
def check_if_structured_tour(self):
    row_last_square = self.row_dimension - 1
    row_second_last = row_last_square - 1
    row_third_last = row_second_last - 1
    col_last_square = self.col_dimension - 1
    col_second_last = col_last_square - 1
    col_third_last = col_second_last - 1
    # List of squares to be checked [(Row squares), (Col squares)]
    to_be_checked = [
        # Top left corner
        [(0, 1), (0, 2), (2, 0), (1, 0)],
        # Top right corner
        [(0, col_third_last), (0, col_second_last), (1, col_last_square), (2, col_last_square)],
        # Bottom left corner
        [(row_last_square, 1), (row_last_square, 2), (row_third_last, 0), (row_second_last, 0)],
        # Bottom right corner
        [(row_last_square, col_third_last), (row_last_square, col_second_last),
         (row_second_last, col_last_square), (row_third_last, col_last_square)],
    ]
    for corner in to_be_checked:
        for i in range(2):
            x_1 = corner[i][0]
            y_1 = corner[i][1]
            x_2 = corner[i + 2][0]
            y_2 = corner[i + 2][1]
            if abs(self.graph[x_1][y_1] - self.graph[x_2][y_2]) != 1:
                print("Not structured")
                return False

    return True
```

Figure 26: Function to check for structured tours implemented in Python

5 Results & Analysis

In terms of speed, the simulation application is much faster compared to the game, as evidenced by Figure 28 and Figure 30 where it took the game almost 16 hours to create just one knight's tour using Backtracking whereas the simulation was able to create 900 identical tours in almost 7 hours with the same algorithm and starting square as the game. On average, the time taken for the simulation to create a tour is 27 seconds, making the game slower than the simulation by a factor of 2100!

When using Wanrsdorff's algorithm, the results are much more sensible. With the game taking 2.44 seconds to find a tour while the simulation took 2.54 seconds to generate 900 tours.

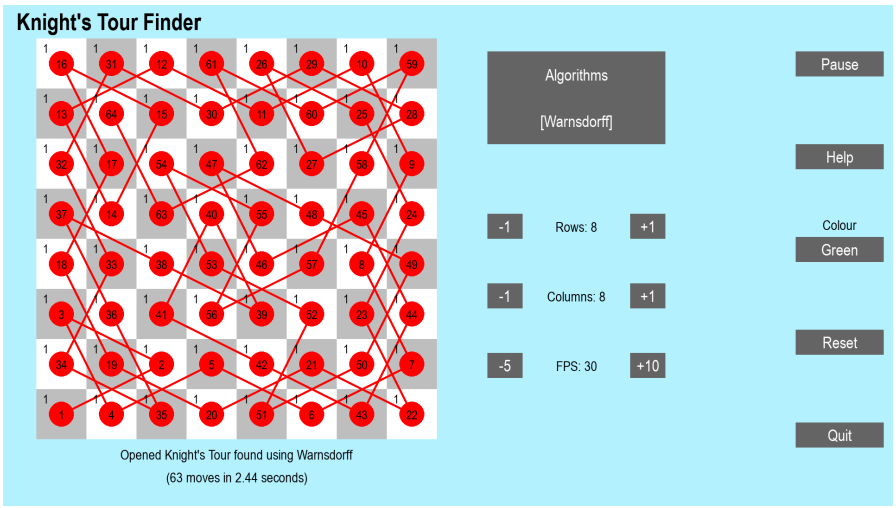


Figure 27: 8×8 Warnsdorff Knight's Tour on the GUI at Square $(7,0)$ in 2.44 seconds at maximum Frames Per Second

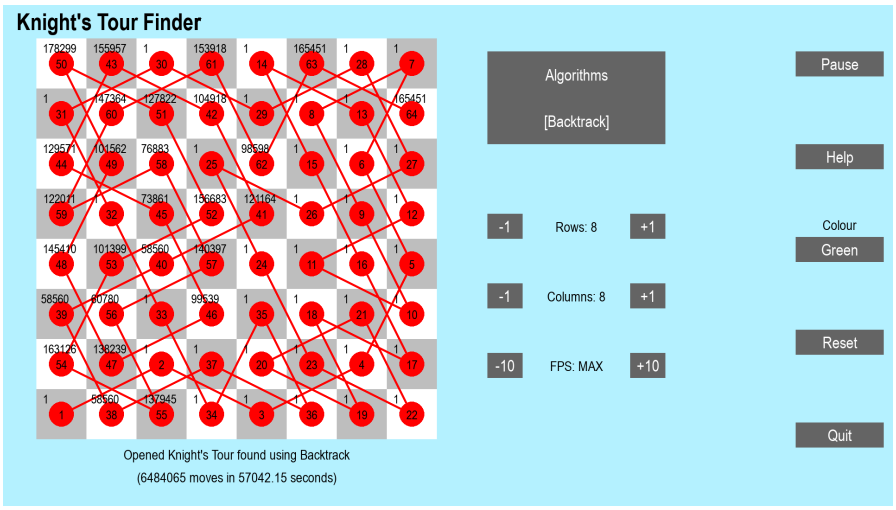


Figure 28: 8×8 Backtrack Knight's Tour on the GUI at Square $(7,0)$ in 57042.15 seconds at maximum Frames Per Second

	position_x	position_y	time	steps
1				
2	7	0	0.003001	64
3	7	0	0.001999	64
4	7	0	0.002002	64
5	7	0	0.002	64
6	7	0	0.002031	64
7	7	0	0.002996	64
8	7	0	0.001999	64
9	7	0	0.004028	64
10	7	0	0.003002	64
11	7	0	0.002	64
12	7	0	0.002003	64
13	7	0	0.003	64
14	7	0	0.003	64
15	7	0	0.001999	64
16	7	0	0.002001	64
17	7	0	0.001991	64
18	7	0	0.003	64
19	7	0	0.001999	64
20	7	0	0.001999	64
21	7	0	0.002001	64
22	7	0	0.001999	64
23	7	0	0.002	64

(a) Top of the list

882	7	0	0.002001	64
883	7	0	0.002999	125
884	7	0	0.002001	64
885	7	0	0.003003	64
886	7	0	0.001997	64
887	7	0	0.002998	64
888	7	0	0.003001	64
889	7	0	0.001999	64
890	7	0	0.003	64
891	7	0	0.002998	64
892	7	0	0.002	64
893	7	0	0.001998	64
894	7	0	0.005001	64
895	7	0	0.003	64
896	7	0	0.003001	64
897	7	0	0.002001	64
898	7	0	0.002003	64
899	7	0	0.004003	64
900	7	0	0.001999	121
901	7	0	0.001999	64
902				
903		Total	2.538525	
904		Average	0.002821	

(b) Bottom of the list

Figure 29: 900 sets of 8×8 Warnsdorff Knight's Tour on the Simulation at Square $(7,0)$

	position_x	position_y	time	steps
2	7	0	18.90662	6484066
3	7	0	34.20085	6484066
4	7	0	38.17615	6484066
5	7	0	38.33595	6484066
6	7	0	41.35998	6484066
7	7	0	39.2712	6484066
8	7	0	39.437	6484066
9	7	0	39.34458	6484066
10	7	0	38.50541	6484066
11	7	0	38.88771	6484066
12	7	0	42.21426	6484066
13	7	0	40.23701	6484066
14	7	0	40.01875	6484066
15	7	0	39.62261	6484066
16	7	0	40.4631	6484066
17	7	0	39.0075	6484066
18	7	0	39.79764	6484066
19	7	0	39.56424	6484066
20	7	0	39.23823	6484066
21	7	0	42.0296	6484066
22	7	0	40.75549	6484066
23	7	0	40.65113	6484066

(a) Top of the list

882	7	0	42.47375	6484066
883	7	0	35.22597	6484066
884	7	0	25.36488	6484066
885	7	0	24.99852	6484066
886	7	0	40.71105	6484066
887	7	0	26.47111	6484066
888	7	0	34.63825	6484066
889	7	0	24.32988	6484066
890	7	0	30.71203	6484066
891	7	0	28.94772	6484066
892	7	0	27.90635	6484066
893	7	0	30.51135	6484066
894	7	0	28.25078	6484066
895	7	0	40.77746	6484066
896	7	0	27.09957	6484066
897	7	0	31.78554	6484066
898	7	0	27.20638	6484066
899	7	0	30.42475	6484066
900	7	0	37.85166	6484066
901	7	0	19.00004	6484066
902				
903		Total	24682.25	
904		Average	27.42472	

(b) Bottom of the list

Figure 30: 900 sets of 8×8 Backtrack Knight's Tour on the Simulation at Square $(7,0)$

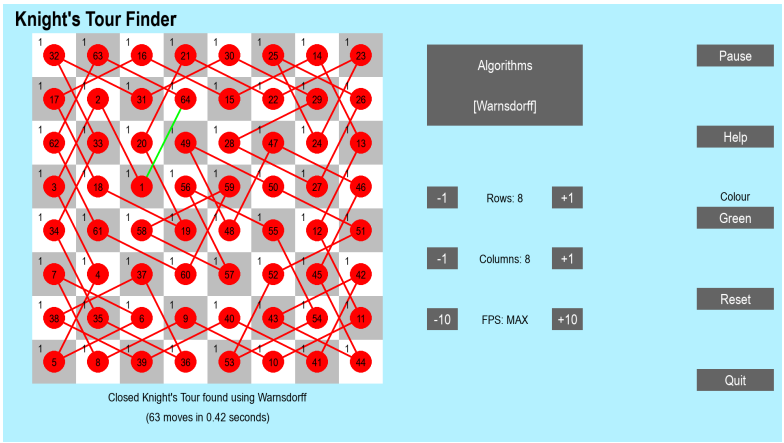


Figure 31: Closed Knight's Tour on 8×8 board using Warnsdorff's algorithm in 0.42 seconds

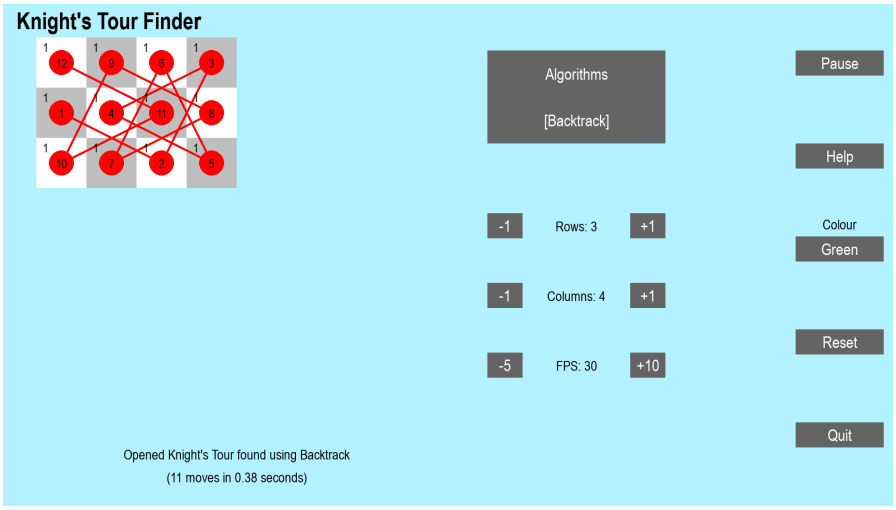


Figure 32: 3×4 Knight's Tour board using Backtrack algorithm on the GUI in 0.42 seconds

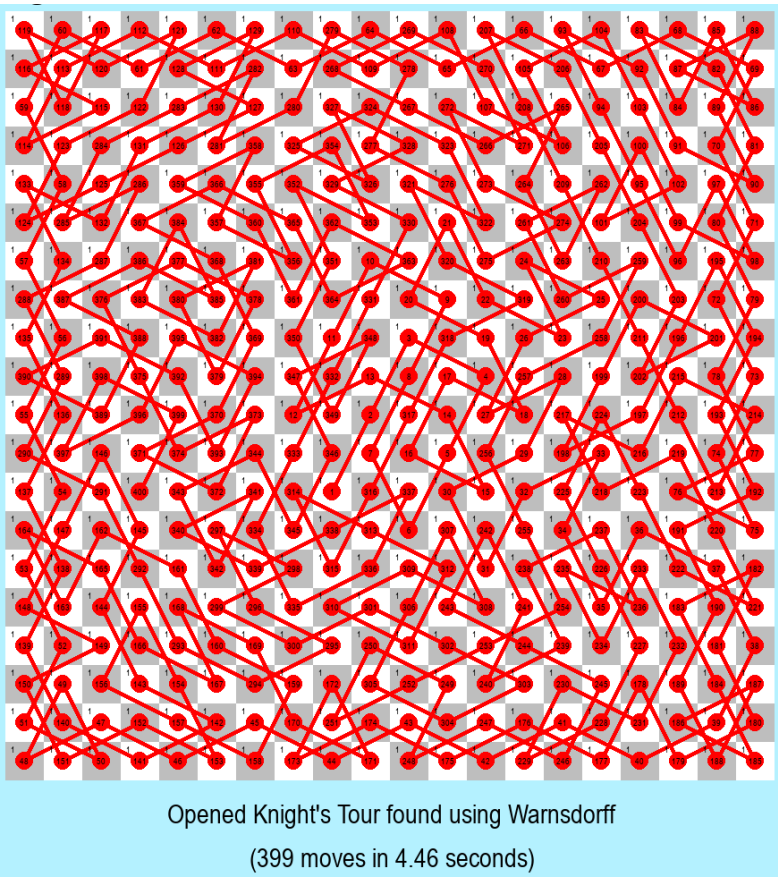


Figure 33: Knight's Tour found on 20×20 board using Warnsdorff's algorithm

6 Conclusion

As a whole, the game and simulation applications work as intended. The game is able to properly display the knight's touring process and the simulation is able to generate more than 1 tour and save them into files. Not only do both applications show that Warnsdorff's algorithm is significantly faster than Backtracking, they also show the impracticality of a GUI for algorithms with large run time complexities like Backtracking. In terms of the adherence to the MoSCoW requirements, almost all have been adhered for the project except the implementation of the Divide and Conquer algorithm which was not done even though it "Could" have.

7 Potential Improvements

Although the outcome of the project was successful and most requirements were fulfilled, there are still many improvements that can be done for the program and features that can be implemented.

One thing that can be improved for the game application is its speed and efficiency for algorithms with long run times. Even with the best case at maximum FPS, it took almost 16 hours just to find a single knight's tour using the Backtrack algorithm compared to the simulation's average time of 25 seconds. A potential solution to this issue is instead of having the game run the touring process by itself, multithreading can be implemented so that while one thread runs the game to display the touring process, a separate thread is run in the background that uses the same algorithm and parameters to find a knight's tour without using the GUI. Once the background thread finds a tour before the game finishes, the game will display a pop-up informing the user that a tour has been found and asks whether the user wishes to skip the touring-process display and show the final result.

For future work, the Divide and Conquer algorithm will be implemented for both the game and simulation. This will require multithreading and pre-made tours created using simulation and saved into files.

References

Ball, W. W. R. (1917). *Mathematical recreations and essays*. Macmillan.

Conrad, A., Hindrichs, T., Morsy, H., and Wegener, I. (1994). Solution of the knight's hamiltonian path problem on chessboards. *Discret. Appl. Math.*, 50(2):125–134.

Euler, L. (1766). Solution d'une question curieuse que ne paroit soumise à aucune analyse. *Mémoires de l'académie des sciences de Berlin*, pages 310–337.

Garey, M. R. and Johnson, D. S. (1979). *Computers and intractability*, volume 174. freeman San Francisco.

Ghosh, D. and Bhaduri, U. (2017). A simple recursive backtracking algorithm for knight's tours puzzle on standard 8×8 chessboard. In *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 1195–1200.

Lin, S.-S. and Wei, C.-L. (2005). Optimal algorithms for constructing knight's tours on arbitrary n× m chessboards. *Discrete applied mathematics*, 146(3):219–232.

McGugan, W. (2007). *Beginning game development with Python and Pygame: from novice to professional*. Apress.

Murray, H. J. R. (1913). *A history of chess*. Clarendon Press.

Parberry, I. (1997). An efficient algorithm for the knight's tour problem. *Discret. Appl. Math.*, 73(3):251–260.

Sandifer, E. (2006). How euler did it. *Washington: Mathematics Association of America*.

Srinath, K. (2017). Python–the fastest growing programming language. *International Research Journal of Engineering and Technology*, 4(12):354–357.

Van Beek, P. (2006). Backtracking search algorithms. In *Foundations of artificial intelligence*, volume 2, pages 85–134. Elsevier.

von Warnsdorf, H. (1823). *Des Rösselsprunges einfachste und allgemeinste Lösung*. Varnhagen.

8 Appendix

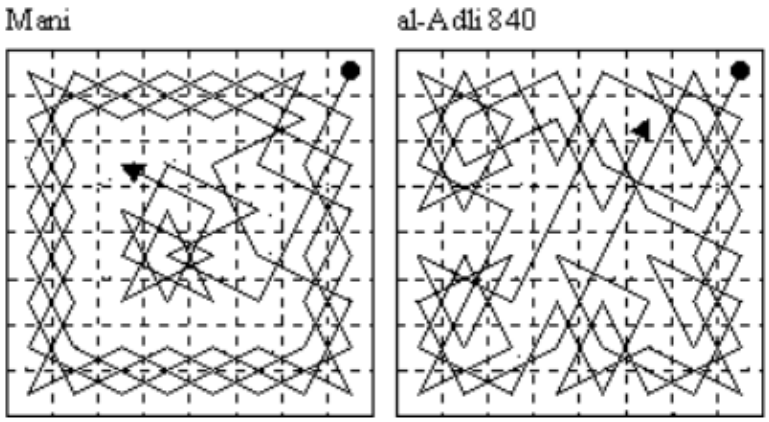


Figure 34: Earliest Knight's Tours

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

(a) Before

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

(b) After

Figure 35: Euler's way of creating a closed tour

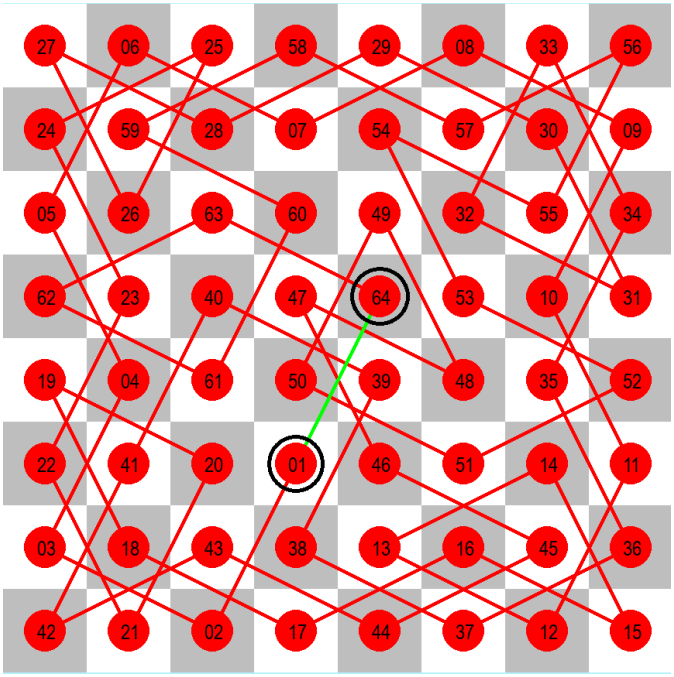


Figure 36: Closed Knight's Tour

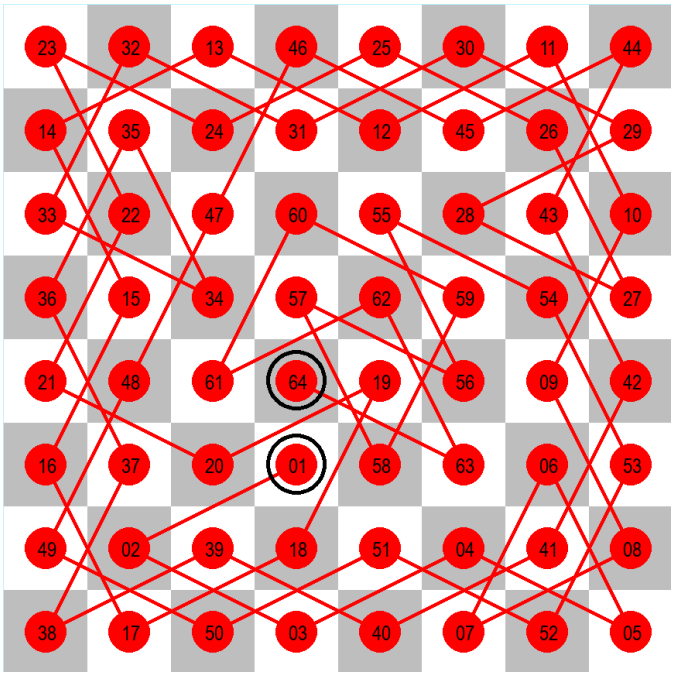


Figure 37: Opened Knight's Tour

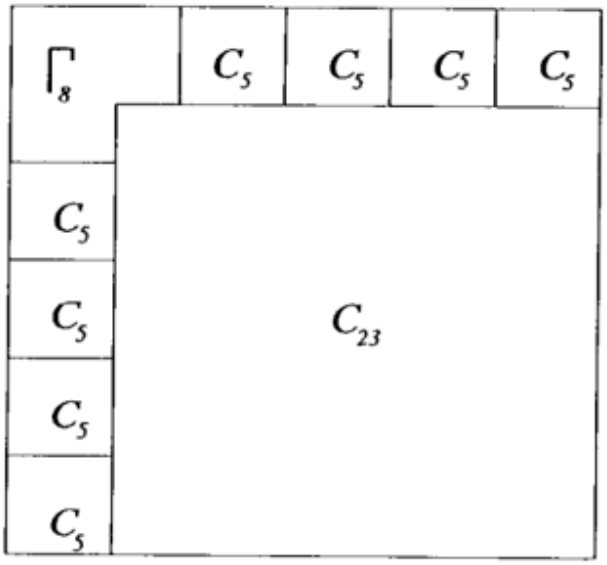


Figure 38: Dividing the board (Conrad et al., 1994)

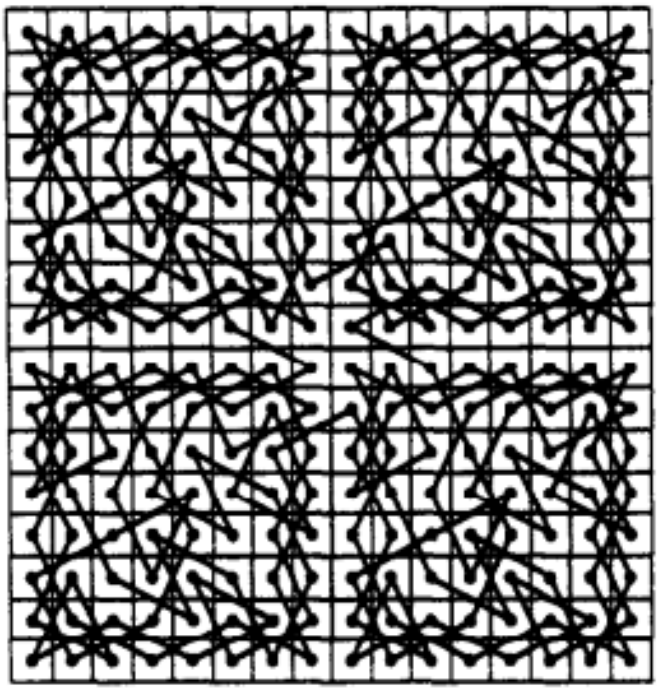


Figure 39: Solving a 16×16 board using Divide and Conquer (Parberry, 1997)