The Knight's Tour Progress Report

Jien Wei, Lim

registration: 100276935

1 Introduction

The Knight's Tour problem was first introduced and analysed by Leonhard Euler (Euler, 1766), which is a sequence of moves made by a knight chess piece visiting every square of a chessboard exactly once. If the knight lands on the final square that is one move away from the starting square, the tour is considered closed (Figure 6), and open (Figure 5) if otherwise.

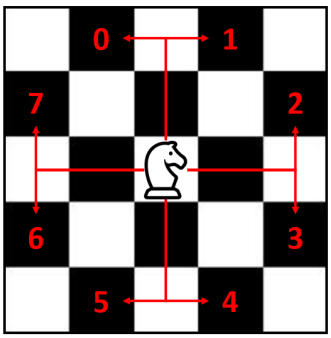

Figure 1: Possible moves that can be made by a knight

2 Aim

The aim of this report is to outline the progress and engagement of the Knight's Tour problem, as well as the issues faced during implementation and future plans for the program.

3 Objectives

The main objective is to create a game-like program that solves the Knight's Tour on an 8×8 chessboard starting from any square selected and displaying the tour on a Graphics User Interface (GUI). The goal at this stage is to complete the implementations of the basic functionalities of the program by the end of the Autumn Semester which are:-

1. Algorithms: Implementing an algorithm responsible for finding the Knight's Tour.
2. GUI: A Graphics User Interface containing an 8×8 chessboard, a knight piece image when placed on the board, and buttons.

3. "Animations": The functions required to display the Knight's Tour process on the GUI's chessboard.

4 Methodology

Due to its data structure handling and extensive libraries, Python was the best candidate to be used as the backbone of the program. Since this will be a game, Python's graphics library for video games, Pygame (McGugan, 2007), will be used to handle the GUI. Learning the library was simple enough thanks to watching videos of someone creating a chess game from scratch. A main point about the library is that when it draws objects onto the display, it is not immediately shown. Instead, a function is needed to be called to update the display and show it on the screen, meaning a continuous main loop is required for the GUI. Figure 2 shows the program's initial operation procedure.

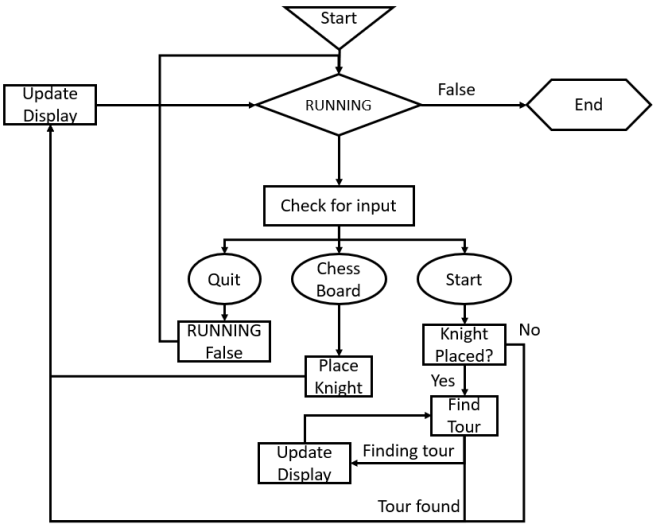

Figure 2: Program's Initial Process Flow

In chess, there are 8 possible L-shaped moves that the Knight is able to make, which can be defined as:-
$(r,c) \in \{(\pm 2, \pm 1), (\pm 1, \pm 2), (\pm 2, \mp 1), (\pm 1, \mp 2)\}$
where r and c are the horizontal and vertical movement in the chessboard respectively. Since this set of moves will only be accessed by the program to move the Knight and not be updated in any way, it will be stored in a tuple data structure.

4.1 Backtracking

Backtracking is an algorithmic method that tries every possible combinations of a solution until one that can solve the problem is found (Civicioglu, 2013). For a chessboard size $n \times n$, the time complexity of the algorithm in the worst case scenario would be $O(8^{n^2})$. This is because the total number of squares in a chessboard is n^2 and for each square, there is a maximum of 8 possible moves to choose from. This makes the algorithm to be considered a brute force method for finding a Knight's Tour.

When looking through online resources, example implementations of it use recursion to find a knight's tour. Instead of following those examples, I chose to implement an iterative Backtrack algorithm to reduce memory usage. This method relies on a stack data structure that acts as the "memory" for the tour. Each element in the stack contains 2 pieces of information: The traversed square and the move index that the Knight uses first if it returns to that square. The algorithm goes as follows:-

```
Store first Knight square and next move in array of logged
moves
While squares are untraversed:
    Get last square element of the move log
    Loop from last square's next to final move index:
        If move is valid:
            Add 1 to last traversed square's move index
            Increase knight's steps by 1
            Add new square and next move index to array
            Break out of loop
    If no valid moves where found:
        Remove last square from move log
        Decrease knight's steps by 1
```

4.2 Warnsdorff Heuristic

Warnsdorff's Heuristic/Rule/Algorithm is a Knight's Tour technique where the knight is moved so that it will move to the square that will have the fewest future moves (von Warnsdorf, 1823). Initially, the plan was to only implement Backtracking for the base functionality. Due to Backtracking's extremely long runtime (Figure 10), I had to im-

plement Warnsdorff's algorithm in order to display a completed Knight's Tour and a full operation of the program that doesn't take hours to complete. Thankfully, this was implemented much quicker compared to Backtracking since there wasn't any need for a "memory". The algorithm goes as follows:-

```
Loop randomly through each possible move:
    Get the new square when using this move
    Count number of valid squares the new square can reach
    If no valid squares are found:
        Reset tour
Use square with smallest count as the next square to land on
Increase knight's steps by 1
```

4.3 GUI

I chose to create a minimalist design for the GUI that contains an 8×8 chessboard with simple buttons. Other than the Knight that uses a Knight piece icon image, the rest of the objects are created using rectangles. The chessboard's coloring is done by calculating the sum of the row and column number of each square and checking whether it is even or odd. Even sum squares are white while odd sum ones are gray. By clicking on any square in the board, a Knight will be placed in that clicked square which serves as the initial position of the Knight and tour. By clicking on the same square again, the Knight will be removed.

There are a total of 5 buttons: "Start" that starts the tour, "Reset" that resets the board to its initial state, "Quit" to quit the game, and 2 more that changes the type of method to find the tour. Only 1 of the 2 method buttons will be shown indicating that the user can click on it to switch to that labeled algorithm to find the tour. This will swap the visibility of the buttons and type of algorithm.

4.4 Animations

When the program is finding a tour, the Knight will be shown moving around the board while leaving number stamps on squares it has traversed that indicates the step count of the tour. I would like to credit Dr Katharina for the suggestion of drawing the Knight

on a square that has a specific value. This method was implemented with some modifications where each square has a default value of -1 and which will allow Pygame to draw a blank square at that position. On squares with a different value, if the square has the highest value, the Knight piece will be drawn on that square. Otherwise, a number stamp is drawn.

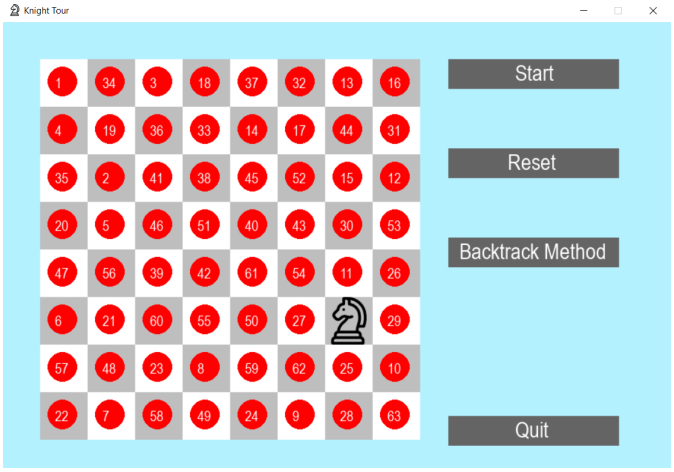

Figure 3: Current Game's GUI

5 Issues and Fixes

If there is a word that can be used to describe the implementation process of this program, "smooth" is not one of them. While there were quite a number of problem during implementation, there was 1 glaring issue that quickly appeared during development which ended up causing overhauls for the program (and sleepless nights). Whenever the program starts running an algorithm to find the tour, it cannot be interrupted, meaning all mouse inputs were ignored and even *quitting* the window was not possible. This was very apparent when running the Backtrack algorithm to find the tour.

To fix it, the program needed to have a new process that prevents the Knight Touring algorithms to have total control of the game when it runs. The way it works now is that the program will have a "state" which determines its operation in every loop. The program simply checks the type of state the game is in and if the program is in a "touring" state, the tour algorithm is run. Instead of continuously looping in the touring

algorithms until a tour is found, the tour algorithm functions now perform one tour step each time it is called. Figure 4 shows the program's new operation procedure.

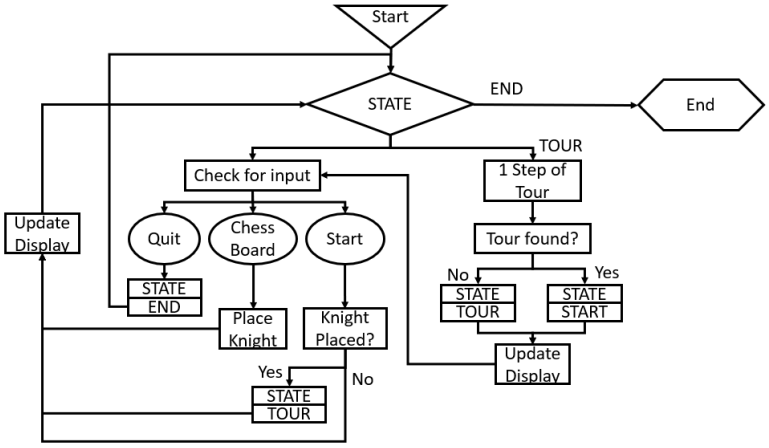

Figure 4: Program's Updated Process Flow

6 Future Plans

Although the basic functions have been successfully implemented, there is still much to be improved upon and even add to the program. In terms of improvements, I am planning on modifying Warnsdorff's Algorithm as the current system redoes the tour finding whenever it hits a dead end. Instead of resetting the board each time a tour fails to be found, it will have a process similar to the iterative Backtracking method where it will log squares that have the same fewest number of futures moves.

For future features, the program will receive a graphical feature to show lines between traversed squares to display the tour more clearly as well as new algorithms for find a tour. There are many different algorithms proposed by different people that I would to try and implement. Although they look quite daunting. I hope to at least implement Ian Parberry's Divide and Conquer algorithm (Parberry, 1997) which can find Knight's Tours on square chessboards larger than 8×8.

References

Civicioglu, P. (2013). Backtracking search optimization algorithm for numerical optimization problems. *Applied Mathematics and computation*, 219(15):8121–8144.

Euler, L. (1766). Memoire de Berlin for 1759. *Berlin, pp. 3lO-337*.

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

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

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

7 Appendix

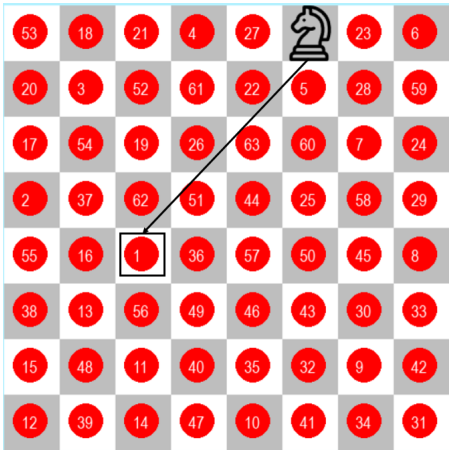

Figure 5: Opened Knight's Tour

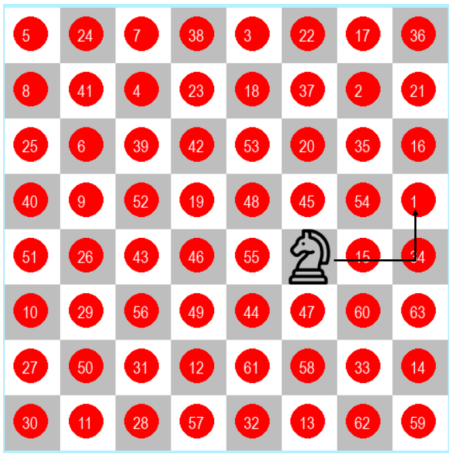

Figure 6: Closed Knight's Tour

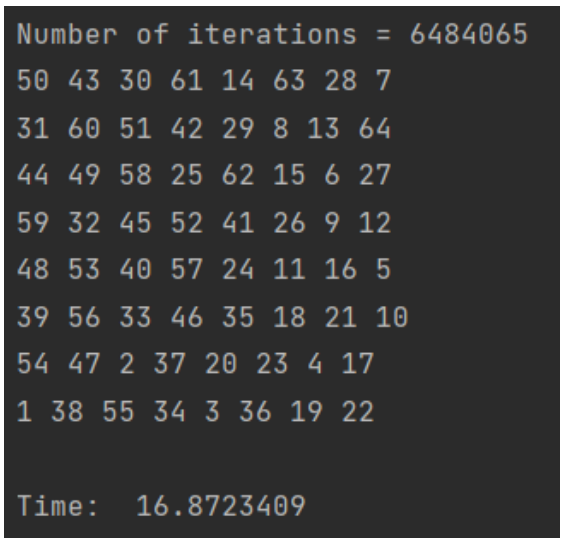

Figure 7: Knight's Tour found using Backtracking starting from bottom left square (6.5 Million Iterations in 16 seconds without being displayed on a GUI)

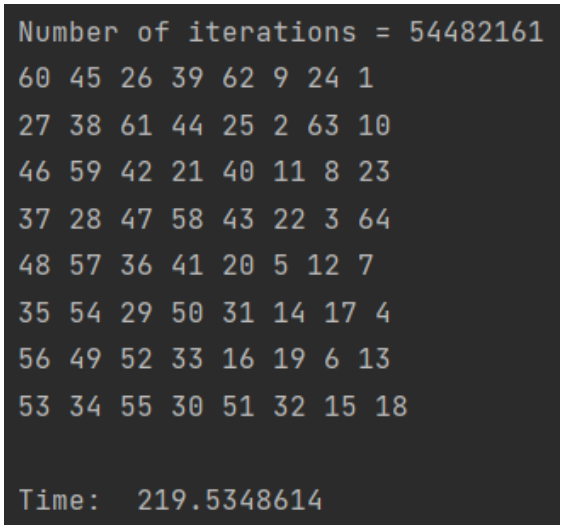

Figure 8: Knight's Tour found using Backtracking starting from top right square (54.5 Million Iterations in 3.5 minutes without being displayed on a GUI)

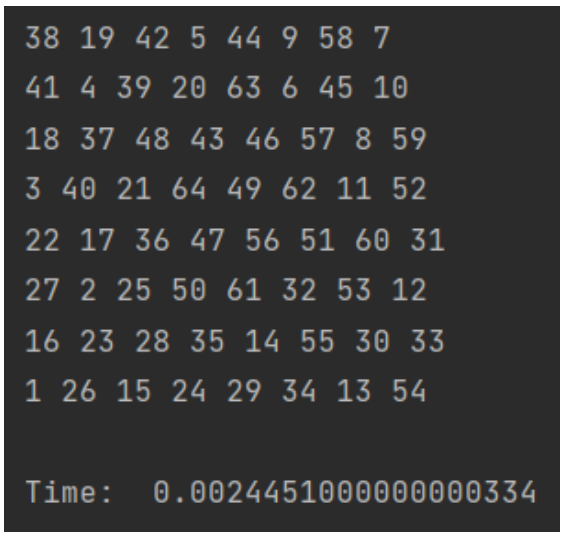

Figure 9: Knight's Tour found using Warnsdorff's Algorithm without being displayed on a GUI

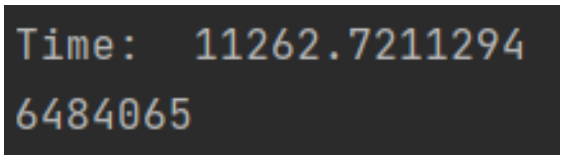

Figure 10: Backtracking using bottom left square with GUI on limitless Frames Per Second (3 HOURS!)

Figure 11: Warnsdorff's Algorithm with GUI on limitless Frames Per Second (3 seconds)

7.1 Previous Work Plan

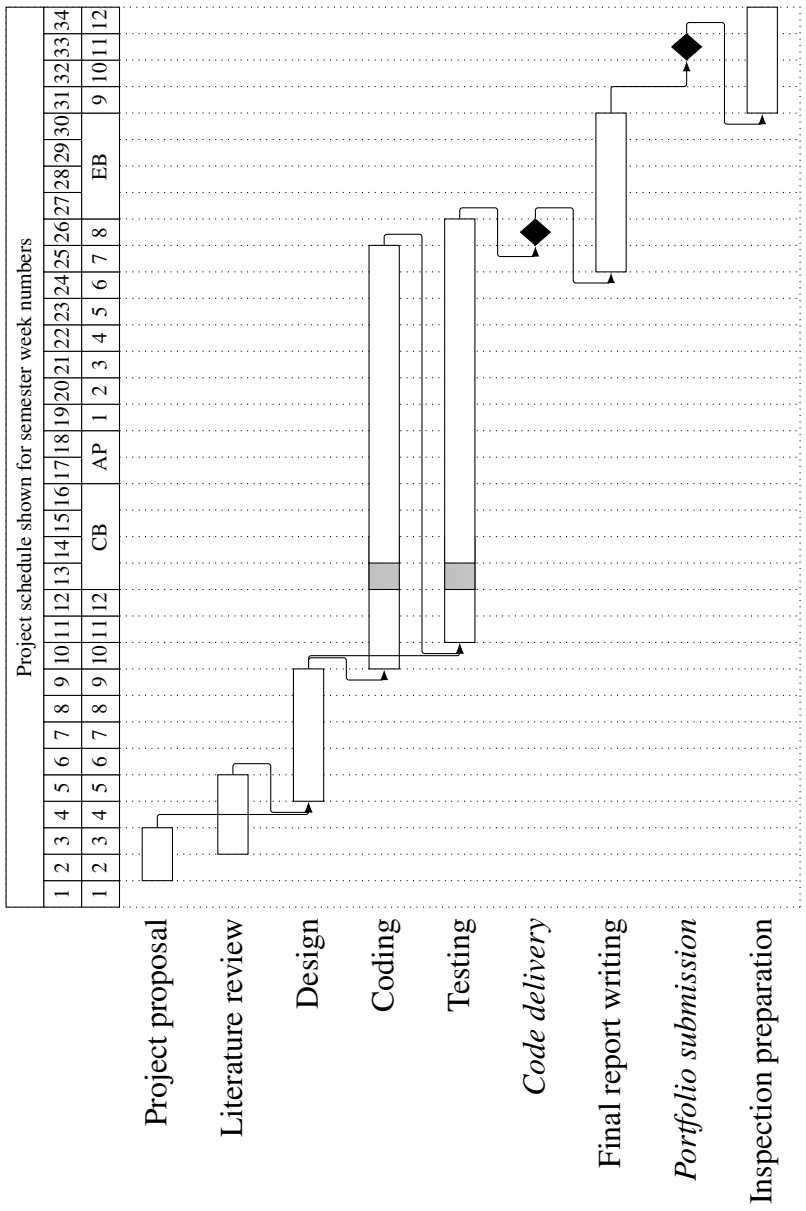

Figure 12: Previous Gantt Chart

7.2 Updated Work Plan

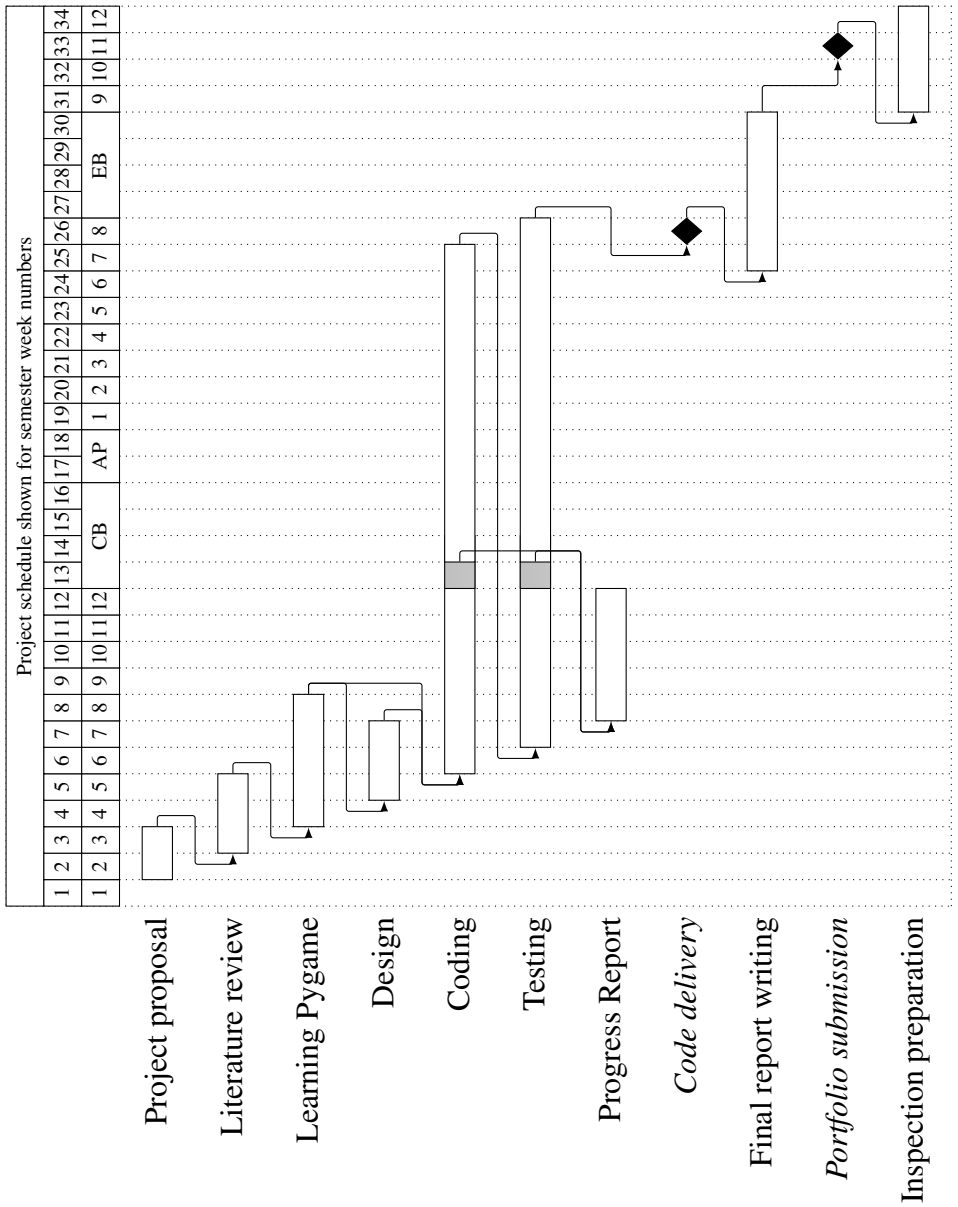

Figure 13: Updated Gantt Chart