Knight's Tour Problem using Backtracking

Anjali Haryani, Mengyuan Liu, Payal Chavan, Yuexin Ma

Abstract

This work explores the application of the backtracking algorithm to solve the Knight's Tour Problem, a well-known challenge in both recreational mathematics and computer science. The Knight's Tour requires navigating a knight on a chessboard such that it visits every square exactly once, following L-shaped movement pattern. Our approach utilizes backtracking, a recursive algorithmic technique that searches for and builds a solution incrementally, backtracking upon reaching a dead-end. This paper details the algorithm's conceptual foundation, provides an illustrative example on a standard chessboard, and presents both theoretical and empirical timing analyses to evaluate performance. Furthermore, we offer comprehensive guidance on how to setup and run the provided program. We conclude with reflections on the project's successes and challenges, offering insights into potential improvements for future explorations.

1 Introduction

The Knight's Tour is a classic chess problem where the goal is to move a knight around the board so that it visits every square exactly once. The knight moves in an L-shape: two squares in one direction and then one square perpendicular, or vice versa. Solving the Knight's Tour involves figuring out a sequence of moves that allows the knight to visit each square without repeating any.

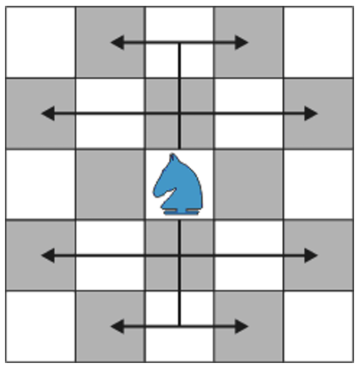


Figure 1: Legal moves for a knight

The problem itself is a specific instance of a more general set of problems known as **Hamiltonian paths and cycles** in graph theory, which involve finding a path or cycle that visits each vertex exactly once. Specifically, the Knight's Tour can be represented as a **Hamiltonian circuit** on an $M \times N$ chessboard graph where each square is a vertex and each legal knight move is an edge.[Swe21]

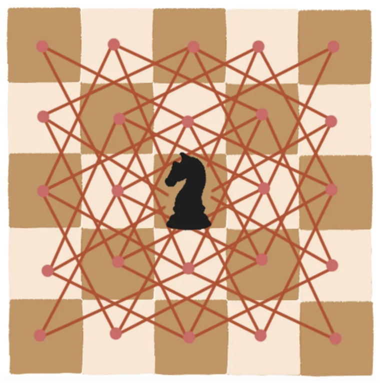


Figure 2: A Knight's Tour graph for a 5×5 board

1.1 Historical Context

The Knight's Tour has a storied history:

- **Early History:** It originated in India during the 9th century as part of recreational mathematics. It spread through the Persian and Arabic worlds, where it was initially studied alongside magic squares.
- **Medieval and Renaissance Europe:** The problem reached Europe by the medieval period, gaining attention from notable mathematicians like Fibonacci in the early 13th century.
- **Euler's Contribution:** In the 18th century, Leonhard Euler significantly advanced the study of the Knight's Tour by applying graph theory, elevating its status in mathematical circles.
- **Modern Computational Approaches:** The advent of computers transformed the Knight's Tour into a common challenge for testing pathfinding and backtracking algorithms. This rich history highlights its enduring appeal in both mathematical theory and algorithmic practice.[Wik24]

Various algorithmic methods can be used to solve the Knight's Tour, each with its unique strategies and efficiencies. These approaches provide insights into both the historical and modern techniques used to tackle this problem:

- **Backtracking:** This recursive approach involves trying out all possible moves from a given position and backtracking when a move doesn't lead to a solution. This method is inefficient for large boards.[GB17]

- **Warnsdorff's Rule (1823):** H.C. Warnsdorff proposed a heuristic to improve the efficiency of finding a Knight's Tour. The rule suggests always moving the knight to the square with the fewest onward moves. This greedy algorithm significantly reduces the search space and often leads to a solution.
- **Divide and Conquer:** Some approaches to the Knight's Tour use divide-and-conquer techniques, breaking down the problem into smaller, more manageable subproblems and combining the solutions.[Wik24]

1.2 Backtracking Algorithm

There are various ways to solve the Knight's Tour Problem. In the programming world, Backtracking can be one answer.
Backtracking is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, and removing those solutions that fail to satisfy the constraints of the problem at any point of time.[Gee24] For the Knight's Tour:

1. **Initialization:** The algorithm starts at a given square, marking it as visited and recording the move sequence.
2. **Exploration:** From the current square, the algorithm explores all possible knight moves that lead to squares that have not yet been visited.
3. **Validation:** For each move, if the square is valid and unvisited, the algorithm moves the knight there, marking the new square as visited and continuing the tour.
4. **Backtracking:** If a move leads to a situation where no further legal moves are possible and the tour is incomplete, the algorithm abandons the current path, returns to the previous square, and tries the next possible move.
5. **Solution Found or Exhausted:** This process repeats until either a complete tour is found (covering all squares exactly once) or all possible paths are exhausted without finding a valid tour.

1.3 Use in Modern Computing

The backtracking approach to the Knight's Tour is a foundation for understanding deeper algorithmic strategies and has implications in modern computing fields like artificial intelligence, robotics, and network routing. It exemplifies how algorithms can handle complex decision-making processes in an efficient manner, even with a significant branching factor and constraints.

1.3.1 Visual Cryptography

The traditional Knight's Tour Problem involves moving a knight on a chessboard in such a way that it visits every square exactly once. For encryption, this concept can be extended to dictate the sequence of pixel or bit manipulation in an image, using the knight's legal moves as a path guide.[SKS15]

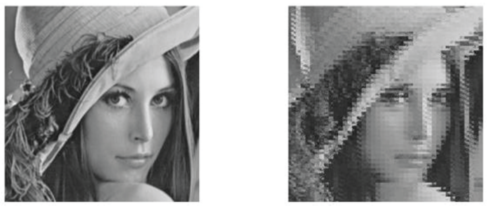


Figure 3: 'Lena' before (left) and after (right) the application of Knight's Tour Scrambler

Image Encryption without Cover Images:

- Unlike traditional visual cryptography, which often requires the use of a cover image to hide the encrypted data, this method uses only the target image. The encryption modifies the image based on the sequence generated by the Knight's Tour, ensuring that the original content is obscured without the need for an additional layer.

2 Implementation and Results

2.1 Objective

Given an $(N \times M)$ chessboard, determine if there exists a sequence of moves for a knight such that it visits every square on the board exactly once. If such a sequence exists, provide the sequence of moves. If no such sequence exists, return an message 'No path found'.

2.2 Algorithm

2.2.1 Pseudo-Code

The following pseudo-code gives an overview on how backtracking works in Knight's Tour.

Algorithm 1 Knight's Tour Algorithm

```
Initialize a chessboard with dimensions $rows \times cols$.
Create an empty $path$ list to store the knight's moves.
Initialize a $board$ matrix with all cells set to $-1$ (indicating unvisited).
Define the possible knight moves as

    $$\text{moves} = [(2,1),(1,2),(-1,2),(-2,1),(-2,-1),(-1,-2),(1,-2),(2,-1)]$$


function IS_VALID_MOVE(r, c)
    Return True if (r, c) is within the board boundaries and unvisited.
end function

function SOLVE(r, c, $move_i$, visualizer)
    Place the knight on cell $(r, c)$ and update the $path$.
    if $move_i$ equals the total number of cells then
        return True                              ▷ Base case: All cells are visited
    end if
    for each possible move in $moves$ do
        Calculate $next_r$ and $next_c$ coordinates.
        if is_valid_move($next_r$, $next_c$) then
            if solve($next_r$, $next_c$, $move_i + 1$, visualizer) then
                return True                      ▷ Solution found
            end if
        end if
    end for
    Backtrack: reset the cell $(r, c)$ and remove it from the $path$.
    return False
end function

function START_TOUR(visualizer=None)
    if solve(0, 0, 0, visualizer) then
        Print "Solution found!" and display the board.
    else
        Print "No path found."
    end if
end function

function PRINT_BOARD
    Display the $board$ matrix.
end function
```

2.2.2 How does Backtracking work in knight tour problem?

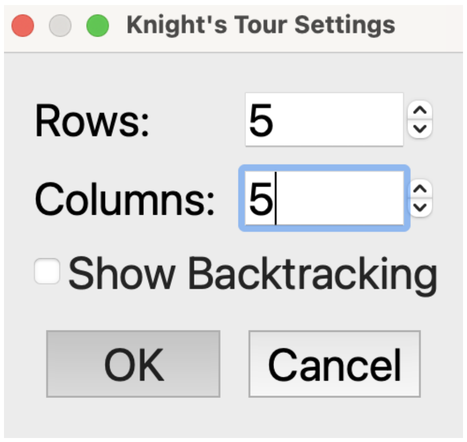
Backtracking is a powerful technique used in solving problems where we explore different paths or choices and backtrack when we encounter a dead end.

1. **Exploration:**
    - Initially, we place the knight on an empty cell of the chessboard.
    - We explore all possible valid moves from that cell (according to knight's movement rules).
    - For each valid move, we recursively explore further by placing the knight on the next cell.
2. **Base Case:**
    - The base case occurs when we have visited all cells on the chessboard (i.e., $move_i$ reaches the total number of cells).
    - The base case checks if the current move number ($move_i$) equals the total number of cells on the chessboard minus 1. (Suppose we have 5×5 chessboard, then we must have $25 - 1 = 24$ moves starting from 0).
    - If this condition is met, we have found a valid solution, and we stop exploring.
3. **Backtracking**
    - If the base case is not met, we backtrack:
        - We undo the last move by resetting the cell value to -1.
        - We remove the last cell from the path.
        - We continue exploring other valid moves from the previous cell.
    - This process continues until we find a solution or exhaust all possibilities.
4. **Optimization (Future Improvements:**
    (a) To improve efficiency, we can prioritize moves based on heuristics (e.g., Warnsdorff's rule) to explore more promising paths first.
    (b) We can also use an additional data structure (such as an array) to keep track of the order in which moves are tried.

2.3 Illustrative Example

2.3.1 Example on a 5×5 Board w/o Showing Backtracking

A typical run starts with the knight at position $(0, 0)$ on an 5×5 board. In this example, we will not use 'Showing Backtracking' feature, thus the backtracking moves are not able to be visualized step-by-step.
Initial Setup:
Consider a 5×5 chessboard. The knight starts at the top-left corner of the board (position $(0, 0)$).


Knight's Tour Settings

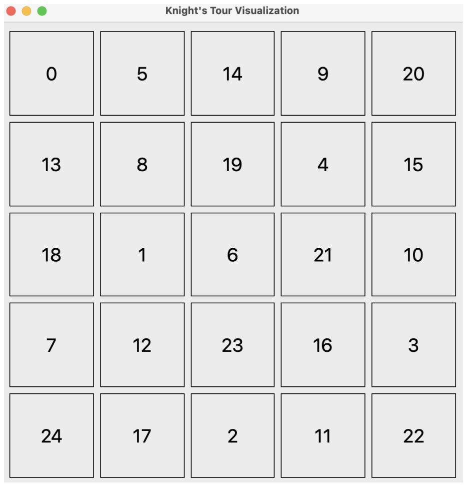
Rows: 5

Columns: 5

Show Backtracking

Figure 4: Input Dialog

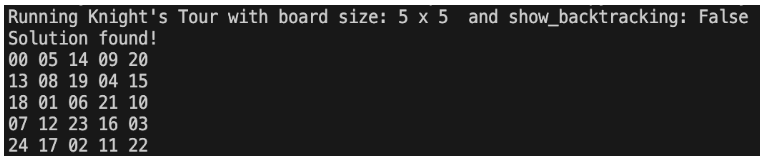
Execution Steps:
A pop-up window will display the chessboard and a valid tour.


Knight's Tour Visualization

0	5	14	9	20
13	8	19	4	15
18	1	6	21	10
7	12	23	16	3
24	17	2	11	22

Figure 5: Chessboard Visualization

Outcome:
The algorithm systematically explores all possible moves, either finding a tour that visits every square once or determining that no such tour is possible. In this case, a valid tour is discovered.

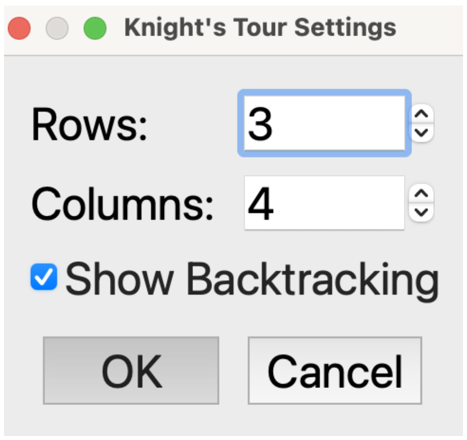


```
Running Knight's Tour with board size: 5 x 5  and show_backtracking: False
Solution found!
00 05 14 09 20
13 08 19 04 15
18 01 06 21 10
07 12 23 16 03
24 17 02 11 22
```

Figure 6: Solution Found

2.3.2 Example on a 3×4 Board with Showing Backtracking

A typical run starts with the knight at position $(0, 0)$ on an 3×4 board. The moves are visualized step-by-step, showing each position the knight moves to, facilitated by PyQt5's GUI capabilities.
Initial Setup:
Consider a 3×4 chessboard. The knight starts at the top-left corner of the board (position $(0, 0)$).



Knight's Tour Settings

Rows: 3

Columns: 4

☑ Show Backtracking

Figure 7: Input Dialog

Execution Steps:
When we choose 'Show Backtracking', a pop-up window will display the chessboard with the specified dimensions with step-by-step moves and all backtracking process.

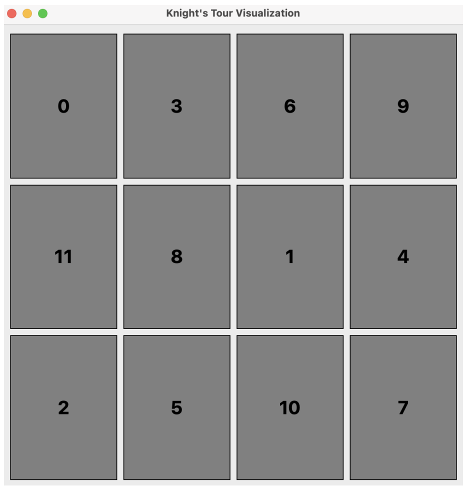
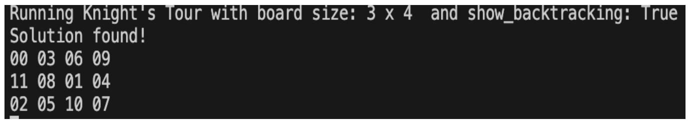


Figure 8: Chessboard Visualization

Outcome:
The algorithm systematically explores all possible moves, either finding a tour that visits every square once or determining that no such tour is possible. In this case, a valid tour is discovered. This example quickly demonstrates that the knight cannot proceed after a couple of moves without revisiting squares, indicating no possible tour. This scenario underscores the necessity for backtracking to explore other potential paths.



```
Running Knight's Tour with board size: 3 x 4  and show_backtracking: True
Solution found!
00 03 06 09
11 08 01 04
02 05 10 07
```

Figure 9: Solution Found

2.4 Theoretical and Empirical Timing Results

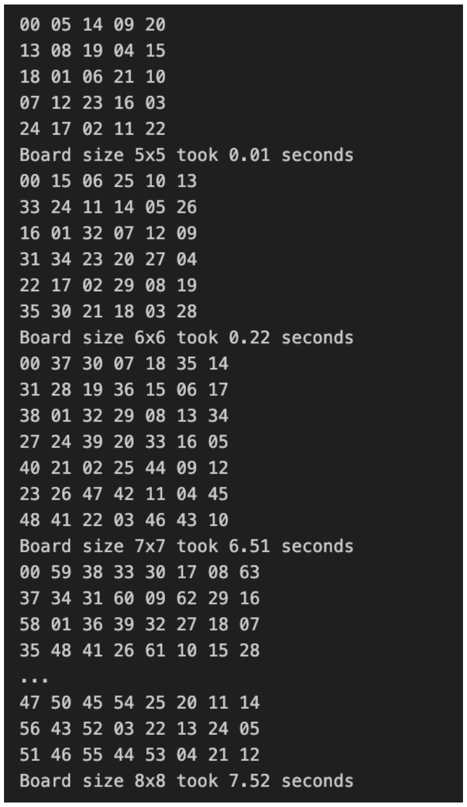
2.4.1 Theoretical Analysis

The Knight's Tour algorithm using backtracking has a **worst-case time complexity** of $(O(8^{N \times M}))$, where N is the number of rows and M is the number of columns on the chessboard. The **best-case time complexity** is $O(N \times M)$ when we find a valid tour quickly without backtracking.
This is because, in the worst case, the algorithm explores all 8 possible moves from each cell, and there are $N \times M$ cells to visit.
The complexity is exponential due to the combinatorial explosion of possible moves as the board size increases.

2.4.2 Experimental Analysis

To perform the timing analysis, we measured the time taken to solve the Knight's Tour problem for various board sizes: $(5 \times 5), (6 \times 6), (7 \times 7)$, and (8×8). Here are the results and graph:

5x5 board:	0.01 seconds
6x6 board:	0.22 seconds
7x7 board:	6.51 seconds
8x8 board:	7.52 seconds



```
00 05 14 09 20
13 08 19 04 15
18 01 06 21 10
07 12 23 16 03
24 17 02 11 22
Board size 5x5 took 0.01 seconds
00 15 06 25 10 13
33 24 11 14 05 26
16 01 32 07 12 09
31 34 23 20 27 04
22 17 02 29 08 19
35 30 21 18 03 28
Board size 6x6 took 0.22 seconds
00 37 30 07 18 35 14
31 28 19 36 15 06 17
38 01 32 29 08 13 34
27 24 39 20 33 16 05
40 21 02 25 44 09 12
23 26 47 42 11 04 45
48 41 22 03 46 43 10
Board size 7x7 took 6.51 seconds
00 59 38 33 30 17 08 63
37 34 31 60 09 62 29 16
58 01 36 39 32 27 18 07
35 48 41 26 61 10 15 28
...
47 50 45 54 25 20 11 14
56 43 52 03 22 13 24 05
51 46 55 44 53 04 21 12
Board size 8x8 took 7.52 seconds
```

Figure 10: Running Time

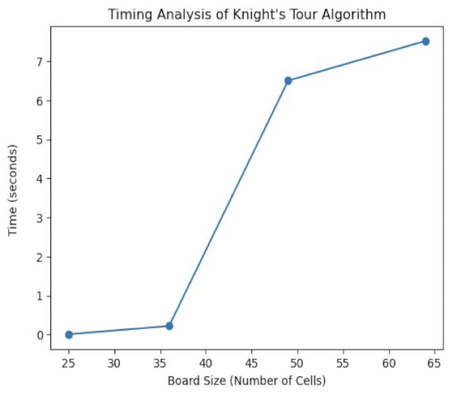


Figure 11: Timing Analysis

- **5x5 Board:** 0.01 seconds. The 5x5 board is relatively small, and the backtracking algorithm can quickly find a solution. There are fewer cells to visit (25 cells), and the search space is manageable, leading to a very short computation time.
- **6x6 Board:** 0.22 seconds. The 6x6 board introduces more cells (36 cells), increasing the search space. The time taken to find a solution increases significantly because the number of possible moves and backtracking steps increases.
- **7x7 Board:** 6.51 seconds. The 7x7 board has 49 cells, further increasing the complexity. The exponential growth in possible paths the knight can take is evident here, as the computation time jumps to several seconds. This indicates that the algorithm is exploring a much larger set of possibilities and performing more backtracking steps.
- **8x8 Board:** 7.52 seconds. The 8x8 board, with 64 cells, represents a standard chessboard. Interestingly, the time taken here (7.36 seconds) is not much greater than that for the 7x7 board. This could be due to specific optimizations or the inherent characteristics of the problem at this size, where certain heuristics or pruning techniques might become more effective.

Exponential Growth:
The results reflect the **exponential nature of the backtracking algorithm**. As the board size increases, the number of possible knight moves grows rapidly. This leads to a significant increase in the time required to explore all potential solutions.
The jump in time from 5×5 to 6×6 (0.01 to 0.22 seconds) and from 6×6 to 7×7 (0.22 to 6.51 seconds) illustrates how quickly the problem becomes more complex. The relatively smaller increase from 7×7 to 8×8 suggests that there might be some specific efficiencies in the algorithm for this particular size or

that the solution path is found earlier in the search process.
You can plot the results with the number of cells on the x-axis and the computation time on the y-axis as graph representation. This will show a steep curve, indicative of exponential growth.

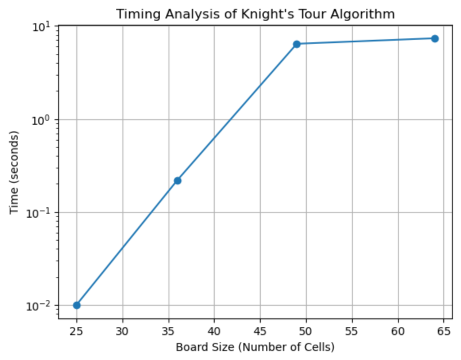


Figure 12: Timing Analysis on a Logarithmic Scale

The above revised graph is plotted on a logarithmic scale, which better illustrates the exponential growth in execution time as the board size increases. The X-axis represents the board size (number of cells), and the Y-axis, plotted on a logarithmic scale, shows the execution time in seconds.

2.5 Technical Details

Python Libraries Used:
The implementation leverages Python due to its excellent support for recursive functions and ease of use for rapid prototyping. Key libraries include:

```
import time
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow, QGridLayout, ...
from PyQt5.QtCore import Qt, QTimer
```

- time: For measuring execution times of the algorithm.
- sys: To interface with the host system for various parameters.
- PyQt5: For graphical user interface components, enabling real-time visualization of the algorithm's progress.

Reasons behind choosing PyQt5 library for visualization:

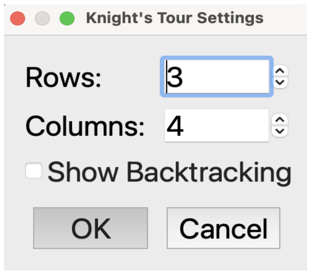
- PyQt5 provides a robust framework for creating GUI applications.
- Users can visually follow the knight's path and see the solution unfold step by step.

- You can design an interactive chessboard interface with buttons representing cells.
- Users can observe the knight's movement and appreciate the algorithm's progress.
- PyQt5 allows you to create a separate panel or log to show the order of visited cells.
- PyQt5's flexibility lets you create an appealing and informative interface.
- PyQt5 has an active community and extensive documentation.

2.6 Running the Program

Instructions on how to run the code:

1. **Input Dialog:**
   - When you run the code, an Input Dialog will appear.
   - In this dialog, you'll need to input the dimensions of the chessboard (number of rows and columns). By default, the values are set to 3×4, but you can adjust them according to your preference.
   - Additionally, you'll find an option to toggle "Show Backtracking". If checked, it will display all the backtracking steps during the solution process.
   - Click the "OK" button to proceed.



Rows: 3
Columns: 4
Show Backtracking

Figure 13: Input Dialog Example

2. **Chessboard Visualization:**
   - After clicking "OK," a new window will pop up.
   - This window will display the chessboard with the specified dimensions.
   - The knight's moves will be shown as a sequence of numbers (representing the order in which cells are visited).

- If you toggled on “Show Backtracking”, you’ll see the backtracking steps highlighted as well.

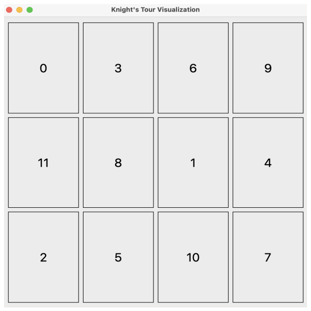
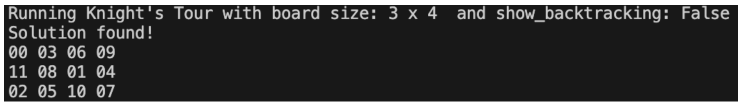
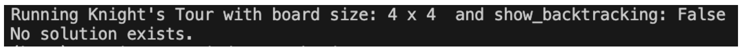


Figure 14: Chessboard Visualization Example

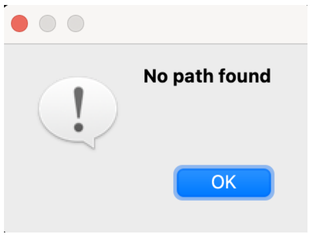
3. **Solution and Output:**
   - If a solution exists (i.e., a valid knight’s tour is found), the console will print “Solution Found.”
   - The final board will be printed, showing the knight’s path.
   - If no solution exists (e.g., for a 4x4 board), a Message Box will appear, indicating “No path found.”



```
Running Knight's Tour with board size: 3 x 4  and show_backtracking: False
Solution found!
00 03 06 09
11 08 01 04
02 05 10 07
```



```
Running Knight's Tour with board size: 4 x 4  and show_backtracking: False
No solution exists.
```



No path found

Figure 15: Two Types of Solution Examples

2.7 Input Formatting

The user is prompted to input the dimensions of the chessboard through a GUI dialogue.

1. The chessboard dimensions (number of rows and columns) must be specified as integers.
2. The minimum allowed value for both rows and columns is 1, and the maximum allowed value for both rows and columns is 20.

3 Project Reflection

Successes and Challenges:

1. **Choice of Visualization Library (PyQt5):** Selecting the right visualization library can indeed be challenging. PyQt5 is a popular choice for creating graphical user interfaces (GUIs) in Python, but it's not the only option. Other libraries like Tkinter, Zelle, wxPython, offer different trade-offs. While working with these other libraries, issues related to rendering windows and dialog boxes arised. Debugging these issues was time-consuming.
2. **Slow Backtracking Steps:** Backtracking algorithms could be time-consuming to visualize step-by-step. Even for smaller dimensions (e.g., 5x5), displaying each move at a reasonable pace can take a significant amount of time.
3. **Edge Cases and Constraints:** Addressing edge cases is essential when implementing the Knight's Tour problem. These edge cases involve irregular board shapes (which need not be an 8x8 square) and ensuring that the knight wraps around the edges (for example, moving from the right edge to the left edge). Implementing boundary conditions and handling these edge cases gracefully can indeed be challenging.
4. **GUI Implementation Complexity:** Developing a graphical user interface (GUI) for visualizing the Knight's Tour requires designing widgets, managing event handling, and ensuring smooth interaction between the user and the algorithm. While PyQt5 offers robust tools for GUI development, mastering its features can indeed be a challenging endeavor.
5. **Knight's Tour Visualization:** When creating visualizations for the Knight's Tour problem, we faced several challenges. These include determining how to display the chessboard, representing the knight's path and each move visually, ensuring smooth transitions between positions, showing backtracking steps when the knight reaches a dead end, and selecting an optimal update rate for user-friendly viewing.

4 Acknowledgements

We would like to express the gratitude to the professor Dr. Bruce Maxwell. His knowledge and teaching have not only enriched our learning experience but also

inspired us to explore complex problem-solving techniques like the backtracking algorithm used in this project.
Our sincere thanks go to all the members of our team: Anjali Haryani, Mengyuan Liu, Payal Chavan, and Yuexin Ma. Each member has shown dedication and effort, contributing unique insights and skills that were crucial for the successful completion of this project.
We would also like to acknowledge the resources that supported our research, including various articles, papers, and tools. Special thanks to the IEEE and other academic journals that provided critical data and frameworks that guided our theoretical and empirical analyses.

References

[GB17] Debajyoti Ghosh and Uddalak Bhaduri. “A Simple Recursive Backtracking Algorithm for Knight’s Tours Puzzle on Standard 8×8 Chessboard”. In: *IEEE Conference Proceedings* (2017).

[Gee24] GeeksforGeeks. *The Knight’s Tour Problem.* `https://www.geeksforgeeks.org/the-knights-tour-problem/`. Apr. 2024.

[SKS15] M. Singh, A. Kakkar, and M. Singh. “Image Encryption Scheme Based on Knight’s Tour Problem”. In: *Procedia Computer Science* 70 (2015), pp. 245–250.

[Swe21] Charlotte Sweeney. “The Knight’s Tour: Finding Some of its Solutions”. In: (2021).

[Wik24] Wikipedia. *Knight’s Tour.* `https://en.wikipedia.org/wiki/Knight's_tour`. Aug. 2024.