Advanced Algorithms and Complexity - CSE245

G3/S5

Presented by:

Omar Montaser Mahmoud - 23P0171

Introduction

This part explores the feasibility of completing a closed knight's tour—a sequence of legal knight moves that visits every square on an n x n chessboard exactly once and ends on a square one move away from the starting point. The problem, rooted in recreational mathematics, is a classic example in algorithm design and graph theory.

We investigate three approaches to solving the problem:

1. A greedy heuristic based on Warnsdorff's Rule,
2. A brute-force backtracking method, and
3. A hybrid algorithm that combines Warnsdorff's heuristic with recursive backtracking.

Each method is evaluated for its performance and reliability across different board sizes, with special focus on how board dimensions affect the success and efficiency of the algorithms.

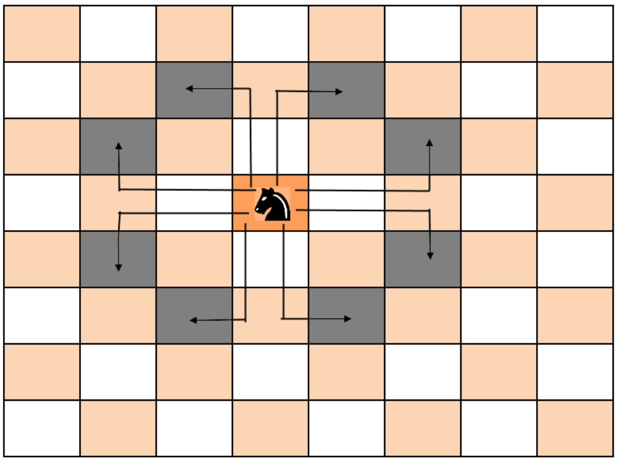


Figure 1

Problem Description

This study addresses the following core questions:

- Can a knight complete a *closed* tour on a standard 8 × 8 board?

- Can the problem be generalized to n × n boards? If so, which values of *n* allow for a closed tour?
- Can a greedy algorithm based on Warnsdorff's Rule efficiently find such a tour?
- What is the performance of this approach across different board sizes and starting positions?

Theoretical Solution

Allen J. Schwenk's seminal 1991 paper, "Which Rectangular Chessboards Have a Knight's Tour?" [1], provides a comprehensive characterization of rectangular boards that admit a closed knight's tour. The core result of the mathematical paper is often paraphrased as follows:

A closed knight's tour exists on an m × n chessboard if and only if:

- m and n are not both odd,
- m ≠ 1, 2, or 4,
- and if m = 3, then n ≠ 4, 6, or 8.

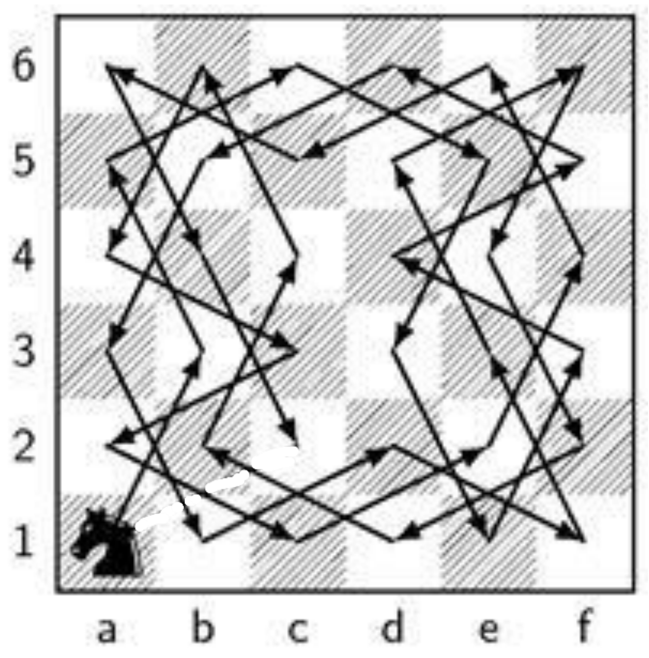
Therefore, we can conclude that for:

- **General n × n Boards**
    - No closed tour exists for n = 1, 2, 3, 4, or 5.
    - A closed tour exists if n ≥ 6 and n is even.
    - **No closed tour exists on odd-sized boards** with n ≥ 5, due to the knight's color-alternating nature (requiring an even number of squares for a loop) .
- **An 8 × 8 Chessboard**

  A closed knight's tour is known to exist on the 8 × 8 board, as established by Schwenk's Theorem.

Assumptions

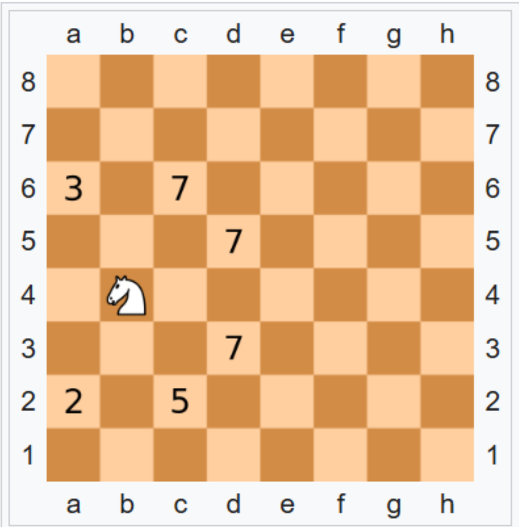
- The knight moves according to standard chess rules: in an L-shape — two squares in one direction and one square in a perpendicular direction.
- The board is a square of size **n × n**.
- Each square must be visited **exactly once** — no repetitions are allowed.
- A **closed tour** ends on a square that is **one legal knight's move away** from the starting square.
- A tour **may not exist** for all n, and success depends on the board size and the knight's movement constraints.
- The algorithm **attempts all possible starting positions** to increase the likelihood of finding a valid closed tour.
- The greedy algorithm is based on **Warnsdorff's Rule**, selecting the next move that leads to the **fewest onward moves** from the candidate position.
- Ties between moves with equal onward degrees are broken **deterministically** by evaluating knight moves in a fixed, pre-defined order.
- **No randomness or backtracking** is used — the algorithm strictly follows local optimality at each step.
- A knight's tour is considered valid if **all n^2 squares are visited exactly once**. For a **closed tour**, the final square must be **one knight move** from the starting square.



Algorithm Design

Base Greedy (Warnsdorff's Rule)

In 1823, H. C. von Warnsdorff introduced a heuristic for solving the knight's tour problem, stating: "Always move the knight to an adjacent, unvisited square with the fewest onward moves. This strategy aims to minimize the risk of the knight becoming trapped by prioritizing moves that lead to squares with limited future options. By doing so, it helps in constructing a complete tour without revisiting any square. While Warnsdorff's rule doesn't guarantee a solution in every case, it significantly improves the chances of finding a complete tour, especially on standard 8×8 chessboards.



Algorithm Steps:

- Start from each possible square (x, y) on the board002E
- At each step, among all legal knight moves from the current square:
    - Choose the one that leads to the **fewest onward valid moves**.
    - If multiple moves have the same onward count, pick the first in the fixed order (no random tie-breaking).
- Continue until:
    - Either all squares are visited (success), or

- No legal move is possible (failure).

- After visiting all squares, check if the final position is a **knight move away from the starting point** (for closed tours).

Pseudocode:

```
function findClosedTourFrom(startX, startY):
    Initialize board with -1 (unvisited)
    Set current position = (startX, startY), mark it as visited
    for i = 1 to n^2 - 1:
        Choose next move with minimum onward moves (Warnsdorff)
        If no such move exists: return failure
        Mark next move as visited
    Check if final square is a knight move from (startX, startY)
    If so, return success
```

Complexity Analysis

- **Number of knight moves per square:** There are at most **8 possible knight moves** (in general). This is a constant factor.

- **At each step:** For each of the 8 possible moves, we need to count the onward moves from the new position. Counting the onward moves involves checking up to 8 possible moves again, which is constant in time.

- **Total steps in the tour:** In the worst case, there are n^2 steps (one for each square on the board).

Thus, the **time complexity** for the greedy approach is:

O(n ^ 2 · 8) = O(n ^ 2)

For space complexity, we need to store a 2D board of size n x n to track visited squares, which requires O(n ^ 2) space.

C++ Implementation:

```
#include <iostream>
#include <vector>
#include <iomanip>

using namespace std;

vector<int> cx = {1, 1, 2, 2, -1, -1, -2, -2};
vector<int> cy = {2, -2, 1, -1, 2, -2, 1, -1};

int N;

bool limits(int x, int y) {
    return x >= 0 && y >= 0 && x < N && y < N;
}

bool isempty(const vector<int>& board, int x, int y) {
    return limits(x, y) && board[y * N + x] < 0;
}

int getDegree(const vector<int>& board, int x, int y) {
    int count = 0;
    for (int i = 0; i < 8; ++i) {
        if (isempty(board, x + cx[i], y + cy[i])) {
            count++;
        }
    }
    return count;
}

bool nextMove(vector<int>& board, int& x, int& y) {
    int minDeg = 9;
    int minIdx = -1;
    int nx, ny;

    for (int i = 0; i < 8; ++i) {
        nx = x + cx[i];
        ny = y + cy[i];
        if (isempty(board, nx, ny)) {
            int deg = getDegree(board, nx, ny);
            if (deg < minDeg) {
                minDeg = deg;
                minIdx = i;
```

```
            }
        }
    }

    if (minIdx == -1) return false;

    nx = x + cx[minIdx];
    ny = y + cy[minIdx];
    board[ny * N + nx] = board[y * N + x] + 1;
    x = nx;
    y = ny;
    return true;
}

void printBoard(const vector<int>& board) {
    const int cellWidth = 4;

    auto printLine = [](int N, int width, char left, char middle, char right,
char fill) {
        cout << left;
        for (int i = 0; i < N - 1; ++i) {
            cout << string(width, fill) << middle;
        }
        cout << string(width, fill) << right << "\n";
    };

    printLine(N, cellWidth, '+', '+', '+', '-');

    for (int i = 0; i < N; ++i) {
        cout << "|";
        for (int j = 0; j < N; ++j) {
            cout << setw(cellWidth) << board[i * N + j] << "|";
        }
        cout << "\n";
        printLine(N, cellWidth, '+', '+', '+', '-');
    }
}

bool isNeighbour(int x, int y, int sx, int sy) {
    for (int i = 0; i < 8; ++i) {
        if ((x + cx[i] == sx) && (y + cy[i] == sy)) {
            return true;
        }
    }
    return false;
}

bool findClosedTourFrom(int sx, int sy) {
```

```
    vector<int> board(N * N, -1);
    int x = sx, y = sy;
    board[y * N + x] = 1;

    for (int i = 0; i < N * N - 1; ++i) {
        if (!nextMove(board, x, y))
            return false;
    }

    if (!isNeighbour(x, y, sx, sy)) {
        return false;
    }

    cout << "Closed knight's tour found starting at (" << sx << ", " << sy <<
"):\n";
    printBoard(board);
    return true;
}

int main() {
    cout << "Enter board size n (n x n): ";
    cin >> N;

    if (N < 1) {
        cout << "Invalid board size.\n";
        return 1;
    }

    for (int sx = 0; sx < N; ++sx) {
        for (int sy = 0; sy < N; ++sy)
            if (findClosedTourFrom(sx, sy)) {
                return 0;
            }
    }

    cout << "No closed knight's tour found for this board size.\n";
    return 0;
}
```

Alternative Approach: Backtracking

The backtracking method systematically explores all possible knight moves from the current position, marking visited squares and backtracking upon reaching dead ends. This exhaustive search ensures that all potential paths are considered.

Pseudocode:

```
function solveKT():
    Initialize board[N][N] with -1
    Set board[0][0] = 0  // Start from top-left corner
    Define moveX[8] = {2, 1, -1, -2, -2, -1, 1, 2}
    Define moveY[8] = {1, 2, 2, 1, -1, -2, -2, -1}
    if solveKTUtil(0, 0, 1, 0, 0, board, moveX, moveY) is true:
        print board
    else:
        print "No solution"
function solveKTUtil(x, y, movei, startX, startY, board, moveX, moveY):
    if movei == N * N:
        if isNeighbor(x, y, startX, startY):
            return true
        else:
            return false
    for k = 0 to 7:
        next_x = x + moveX[k]
        next_y = y + moveY[k]
        if isSafe(next_x, next_y, board):
            board[next_x][next_y] = movei
            if solveKTUtil(next_x, next_y, movei + 1, startX, startY, board, moveX,
moveY):
                return true
            board[next_x][next_y] = -1  // Backtrack


    return false
```

Explanation:

- Starts from a given cell (usually top-left).
- Explores all 8 knight moves recursively.
- If a move leads to a dead-end, it backtracks.
- Base case: All squares are visited.

Time Complexity

Worst-case time complexity: O(8 ^ (n ^ 2)); at each of the n^2 squares, the knight can make up to 8 moves, leading to an exponential number of possible paths.

Space Complexity

Auxiliary space: O(n^2); uses a 2D array to track visited squares and a recursion stack.

Pros

- Guaranteed to find a solution if one exists.
- Straightforward to implement.

Cons

- Highly inefficient for larger boards due to exponential time complexity.

Final Enhancement: Hybrid Approach (Warnsdorff's Heuristic + Backtracking)

This approach combines Warnsdorff's heuristic with backtracking. Initially, it selects the next move based on the square with the fewest onward moves (Warnsdorff's Rule). If a dead end is encountered, the algorithm backtracks to explore alternative paths.

Pseudocode:

```
function solveHybridKT():
    Initialize board[N][N] with -1
    Set startX = sx, startY = sy  // Starting position (random or fixed)
    Set board[startX][startY] = 0
    Define moveX[8] = {2, 1, -1, -2, -2, -1, 1, 2}
    Define moveY[8] = {1, 2, 2, 1, -1, -2, -2, -1}
    if hybridKTUtil(startX, startY, 1, startX, startY, board, moveX, moveY):
        print board
    else:
        print "No solution"
function hybridKTUtil(x, y, movei, startX, startY, board, moveX, moveY):
    if movei == N * N:
        if isNeighbor(x, y, startX, startY):
            return true
        else:
            return false
    next_moves = getSortedMoves(x, y, board, moveX, moveY)

    for each (nx, ny) in next_moves:
        board[nx][ny] = movei
        if hybridKTUtil(nx, ny, movei + 1, startX, startY, board, moveX, moveY):
            return true
        board[nx][ny] = -1  // Backtrack
    return false
```

Explanation

- Like backtracking, but it orders the knight's possible next moves using **Warnsdorff's heuristic**.
- **Warnsdorff's rule**: Choose the next square with the **fewest onward moves**.
- This greatly reduces unnecessary exploration.
- Still backtracks when stuck but often avoids deep recursion.

Time Complexity

Average-case time complexity: O(n ^ 3) ; Warnsdorff's heuristic significantly reduces the number of paths to explore, leading to a polynomial time complexity in practice.

Worst-case time complexity: Still O(8 ^ (n ^ 2)) In the worst case, the algorithm may need to explore all possible paths, similar to the pure backtracking approach.

Space Complexity

Auxiliary space: O(n ^ 2) Similar to the backtracking approach, it uses a 2D array to track visited squares and a recursion stack.

Pros:

- Much more robust
- Fast due to greedy bias
- Recovers from dead-ends using backtracking

Cons:

- More complex implementation
- Still costly on very large boards

Sample Output for Different Cases of n

In this section, we will display the outputs of the Greedy Approach (Warnsdorff's Rule) for different values of n. These outputs illustrate the behaviour of the algorithm for various scenarios:

Case 1: n < 6 (No closed knight's tour possible)

For board sizes n = 1, 2, 3, 4, and 5, *no valid closed knight's tour exists.*

```
Enter board size n (n x n): 3
No closed knight's tour found for this board size.
```

Case 2: Odd n > 6 (Closed knight's tour does not exist)

For odd-sized boards greater than 6 (e.g., n = 7, 9, 11), closed knight’s tours are not possible due to the knight’s alternating color constraint.

```
Enter board size n (n x n): 7
No closed knight's tour found for this board size.
```

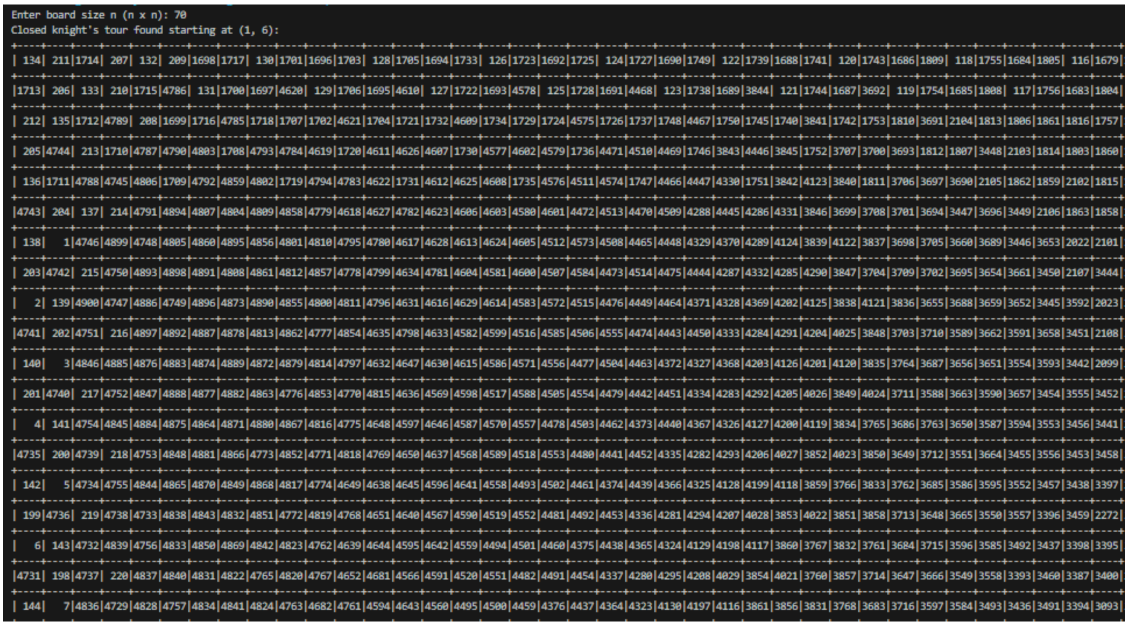
Case 3: Even n > 6 (Closed knight's tour exists)

For even-sized boards greater than 6 (e.g., n = 8, 10, 12), closed knight's tours are theoretically possible, and the algorithm should succeed in finding them.

```
Enter board size n (n x n): 8
Closed knight's tour found starting at (1, 0):
+----+----+----+----+----+----+----+----+
|  14|   1|  30|  63|  16|  11|  34|  61|
+----+----+----+----+----+----+----+----+
|  29|  42|  15|  12|  59|  62|  17|  10|
+----+----+----+----+----+----+----+----+
|   2|  13|  64|  31|  56|  33|  60|  35|
+----+----+----+----+----+----+----+----+
|  41|  28|  43|  58|  49|  54|   9|  18|
+----+----+----+----+----+----+----+----+
|  24|   3|  40|  55|  32|  57|  36|  51|
+----+----+----+----+----+----+----+----+
|  27|  44|  25|  48|  53|  50|  19|   8|
+----+----+----+----+----+----+----+----+
|   4|  23|  46|  39|   6|  21|  52|  37|
+----+----+----+----+----+----+----+----+
|  45|  26|   5|  22|  47|  38|   7|  20|
+----+----+----+----+----+----+----+----+
```

Case 4: Large even n (e.g., n = 70 or larger)

For large even-sized boards, the algorithm can start to show slow performance due to the large search space.



.....

:

Case 5: Too Large n (e.g., n = 1000)

For very large boards, like n = 1000, the greedy approach becomes computationally expensive and impractical due to the growing number of possibilities. It becomes too slow to complete in a reasonable time.

Conclusion

The greedy algorithm based on Warnsdorff's Rule demonstrates its effectiveness for relatively small and even-sized boards (e.g., n = 8), where it can quickly find a closed knight's tour. However, for smaller boards where no tour is possible (n < 6), or for large odd-sized boards (n > 6), it fails to find a valid tour due to the inherent constraints of knight movement. Furthermore, when the board size increases significantly (e.g., n = 70 or larger), the greedy approach becomes slower and struggles with larger search spaces.

In such cases, using a **hybrid approach** that combines the greedy Warnsdorff's heuristic with backtracking **would vastly improve performance**. Backtracking allows the algorithm to explore alternative paths when a dead-end is encountered, recovering from failed attempts.

This enhancement ensures that the knight can find a valid closed tour even on larger and more complex boards (like n = 1000), which would otherwise be impractical using the pure greedy method.

Thus, for larger values of n, incorporating backtracking into the approach is essential to ensure that the algorithm completes successfully within a reasonable time and finds a valid closed knight's tour for all possible board sizes.

References:

[1] Schwenk, Allen. (1991). Schwenk, A.J.: Which rectangular chessboards have a knight's tour? Math. Mag. 64, 325-332. Mathematics Magazine. 64. 10.2307/2690649.

[2] https://www.geeksforgeeks.org/the-knights-tour-problem/

[3] https://en.wikipedia.org/wiki/Knight%27s_tour

[4] https://www.chess.com/blog/Lord_Hammer/the-knights-tour-problem-chess-and-math-combined