A

CIA REPORT

ON

THE KNIGHT'S TOUR PROBLEM USING BACKTRACKING

Submitted By:

ROLL NO	NAME	PRN NO.
01	TANMAY ACHARE	UCS22M1139
09	AKHILESH A.U.	UCS22M1009
15	JAGRUTI BANGAR	UCS22F1015
20	NIKITA BHOGE	UCS22F1020

Subject:Design and Analysis Of Algorithm

In Academic Year: 2024-25

Department of Computer Engineering

Sanjivani College of Engineering,Kopargaon

(An Autonomous Institute)

Guided By:

Prof. P.B.Dhanwate

Sanjivani College of Engineering, Kopargaon

CERTIFICATE

This is certify that

ROLL NO	NAME	PRN NO.
01	TANMAY ACHARE	UCS22M1139
09	AKHILESH A.U.	UCS22M1009
15	JAGRUTI BANGAR	UCS22F1015
20	NIKITA BHOGE	UCS22F1020

(T.Y.Computer)

Has successfully completed their CIA report on

THE KNIGHT'S TOUR PROBLEM USING BACKTRACKING

Towards the partial fulfilment of

Bachelor's Degree in Computer Engineering

During the academic year 2024-25

Prof P.B.Dhanwate
[Guide]

Dr.D.B. Kshirsagar
[H.O.D. Comp Engg]

Dr.A.G.Thakur
[Director]

CONTENTS

1. PROBLEM STATEMENT

Problem Statement:

The Knight's Tour problem is a classical problem in computer science and mathematics, which uses backtracking to find a solution. The objective is to determine a sequence of moves for a knight on an N×N chessboard such that the knight visits every square exactly once. The knight starts at the first block of the board, and it must move according to the rules of chess (L-shaped moves). The solution should print the order in which each square is visited.

Description:

1. Chessboard Setup:
    - N×NN \times NN×N grid where each cell shows the knight's visiting order.
2. Knight’s Movement:
    - Moves in an L-shape (two steps in one direction, one perpendicular).
    - Up to 8 possible moves, staying within bounds and avoiding revisits.
3. Algorithm:
    - Use backtracking to explore moves.
    - Add valid moves to the path; backtrack if no further moves are possible.

2. PROBLEM PRE-REQUISITE

Input Format:

1. **N**: Size of the chessboard (e.g., N=8 for an 8x8 chessboard).
2. **Start Position:** Typically (0,0), but the algorithm supports any starting point.

Output Format:

1. A matrix where each cell represents the order in which the Knight visits the square.
2. If a solution exists, the filled matrix is returned; otherwise, "No Solution" is printed.

0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

3. REQUIREMENT ANALYSIS

Understanding the Problem:

The problem requires exploring all possible sequences of moves while ensuring:

1. The Knight stays within bounds.
2. No square is visited more than once.

Why Use Backtracking?

1. Backtracking ensures all valid solutions are explored systematically.
2. Infeasible paths are abandoned early, saving computation.

Real-Life Applications:

- Robotics: Programming movement patterns on a grid.
- Video Games: AI for chess programs or similar games.
- Logistics: Path optimization in warehouses.

4. SOLVED EXAMPLE

Solved Example: Knight's Tour on a 5x5 Board

Problem Statement:

We want to find a tour for a knight starting from the top-left corner (0, 0) on a 5x5 chessboard. The knight must visit every cell exactly once.

Step-by-Step Solution:

1. Initialization:

Board size N = 5.

Knight starts at position (0, 0).

Initial board setup:

[[0, -1, -1, -1, -1],

[-1, -1, -1, -1, -1],

[-1, -1, -1, -1, -1],

[-1, -1, -1, -1, -1],

[-1, -1, -1, -1, -1]]

2. First Move:

The knight is at (0, 0). It has 8 possible moves, but on a 5x5 board, only valid moves are considered.

Let's choose the first valid move: (2, 1).

Mark this cell as visited with move number 1:

[[0, -1, -1, -1, -1],

[-1, -1, -1, -1, -1],

[-1, 1, -1, -1, -1],

[-1, -1, -1, -1, -1],

[-1, -1, -1, -1, -1]]

3. Second Move:

From (2, 1), the knight chooses another valid move. For example, (4, 2).

Mark the board:

[[0, -1, -1, -1, -1],

[-1, -1, -1, -1, -1],

[-1, 1, -1, -1, -1],

[-1, -1, -1, -1, -1],

[-1, -1, 2, -1, -1]]

4. Third Move:

From (4, 2), the knight selects a new move (3, 4).

Update the board:

[[0, -1, -1, -1, -1],

[-1, -1, -1, -1, -1],

[-1, 1, -1, -1, -1],

[-1, -1, -1, -1, 3],

[-1, -1, 2, -1, -1]]

5. Continuing the Tour:

The knight continues selecting valid moves and updating the board. Here's the progression:

[[0, 7, 16, 11, 22],

[15, 20, 9, 18, 13],

[6, 17, 8, 21, 10],

[19, 14, 23, 12, 5],

[24, 3, 2, 1, 4]]

6. Tour Completion:

The knight successfully visited all cells of the board without revisiting any. The numbers represent the order in which the knight moved.

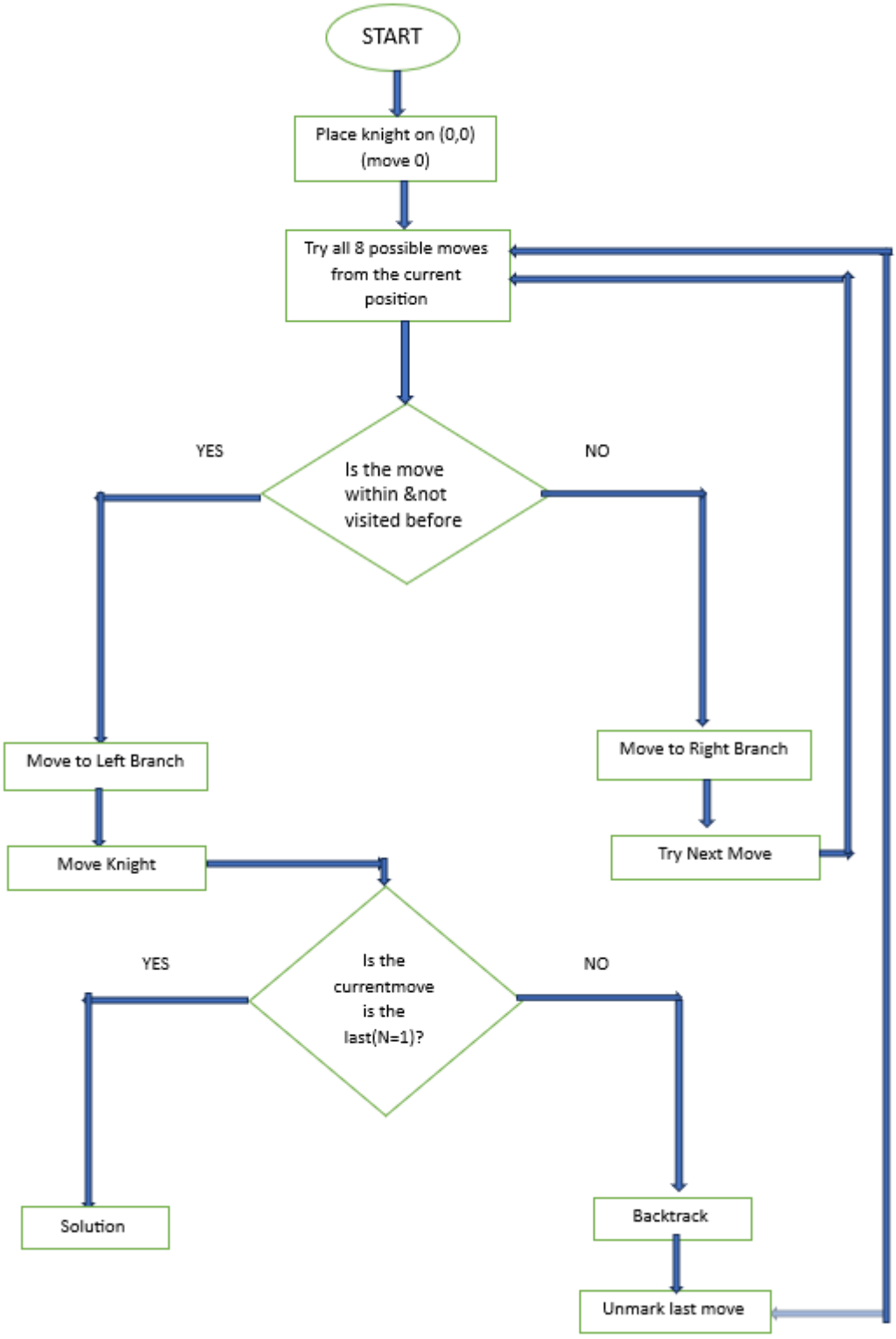
Starting from (0, 0) and ending at (4, 4).

5. CONTROL ABSTRACTOIN AND ALGORITHM

Steps of the Algorithm

1. **Initialization**:
    - Create an N×NN \times NN×N chessboard initialized with -1 to indicate unvisited squares.
    - Define possible knight moves as a list of coordinate offsets: moves=[(2,1),(1,2),(−1,2),(−2,1),(−2,−1),(−1,−2),(1,−2),(2,−1)]\text{moves} = [(2, 1), (1, 2), (-1, 2), (-2, 1), (-2, -1), (-1, -2), (1, -2), (2, -1)]moves=[(2,1),(1,2),(−1,2),(−2,1),(−2,−1),(−1,−2),(1,−2),(2,−1)]
2. **Starting the Tour**:
    - Place the knight at the starting position (0,0)(0, 0)(0,0) and mark it as step 0.
    - Call the recursive function to explore the knight's path.
3. **Recursive Function** (solveKnightTour):
    - If all squares are visited, return True.
    - Otherwise, for each possible move:
      a. Calculate the next position using the current position and the move offset.
      b. Check if the move is valid (within bounds and unvisited).
      c. If valid, mark the square with the current step and recurse.
      d. If the recursion fails, backtrack by resetting the square to -1.
4. **Backtracking**:
    - If no valid moves are possible, backtrack to the previous step and explore other moves.
5. **Completion**:
    - If a solution is found, print the chessboard matrix showing the knight's path.
    - If no solution exists, output "No solution exists."

6. FLOWCHART


START
Place knight on (0,0)
(move 0)
Try all 8 possible moves
from the current
position
Is the move
within &not
visited before
YES
NO
Move to Left Branch
Move Knight
Move to Right Branch
Try Next Move
Is the
currentmove
is the
last(N=1)?
YES
NO
Solution
Backtrack
Unmark last move

7. EXPLAINATION OF ALGORITHMIC STEPS OR CODE OF ENTIRE PROGRAM

Checking Valid Moves:

```
bool isValid(int x, int y, int board[N][N]) {
   return (x >= 0 && x < N && y >= 0 && y < N && board[x][y] == -1);
}
```

Explanation: This function checks if the knight's move (x, y) is within the board limits and if the cell has not been visited.

2. Recursive Backtracking Function:

```
bool solveKTUtil(int x, int y, int movei, int board[N][N], int xMove[], int yMove[]) {
   if (movei == N * N)
      return true;
   for (int i = 0; i < 8; i++) {
      int next_x = x + xMove[i];
      int next_y = y + yMove[i];
      if (isValid(next_x, next_y, board)) {
         board[next_x][next_y] = movei;
         if (solveKTUtil(next_x, next_y, movei + 1, board, xMove, yMove))
            return true;
         board[next_x][next_y] = -1;  // Backtracking
      }
   }
   return false;
}
```

Explanation: This function tries all 8 possible knight moves recursively. It backtracks if no valid moves are left.

3. Initialization and Starting the Knight's Tour:

```
bool solveKT() {
   int board[N][N];
   int xMove[8] = {2, 1, -1, -2, -2, -1, 1, 2};
   int yMove[8] = {1, 2, 2, 1, -1, -2, -2, -1};
   board[0][0] = 0;
```

```
    if (!solveKTUtil(0, 0, 1, board, xMove, yMove)) {
        cout << "Solution does not exist" << endl;
        return false;
    }
}
```

Explanation: This part initializes the board and knight's moves, then calls the recursive function to find the solution.

```
#include <iostream>
using namespace std;
// Size of the chessboard
#define N 8
// Function to check if a move is valid
bool isValid(int x, int y, int board[N][N]) {
    return (x >= 0 && x < N && y >= 0 && y < N && board[x][y] == -1);
}
// Utility function to solve the Knight's Tour problem using backtracking
bool solveKTUtil(int x, int y, int movei, int board[N][N], int xMove[], int yMove[]) {
    if (movei == N * N)
        return true;
    // Try all 8 possible moves from the current coordinate (x, y)
    for (int i = 0; i < 8; i++) {
        int next_x = x + xMove[i];
        int next_y = y + yMove[i];
        if (isValid(next_x, next_y, board)) {
            board[next_x][next_y] = movei;
            // Recursively check for the next move
            if (solveKTUtil(next_x, next_y, movei + 1, board, xMove, yMove))
                return true;
            // Backtracking
            board[next_x][next_y] = -1;
        }
    }
    return false;}
```

```
// Function to solve the Knight's Tour problem
bool solveKT() {
    int board[N][N];
    // Initialize the chessboard with -1
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            board[i][j] = -1;
    // Define the knight's possible moves
    int xMove[8] = {2, 1, -1, -2, -2, -1, 1, 2};
    int yMove[8] = {1, 2, 2, 1, -1, -2, -2, -1};
    // Starting position
    board[0][0] = 0;
    // Start from (0, 0) and explore all moves
    if (!solveKTUtil(0, 0, 1, board, xMove, yMove)) {
        cout << "Solution does not exist" << endl;
        return false;
    } else
        printSolution(board);
    return true;
}
// Function to print the solution matrix
void printSolution(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            cout << board[i][j] << " ";
        cout << endl; } }
// Driver code
int main() {
    solveKT();
    return 0;}
```

8. TIME AND SPACE COMPLEXITY

Time Complexity:

The time complexity of the Knight's Tour problem depends on the size of the chessboard (N× N) and the backtracking approach.

1. Possible Moves per Position:
    - The knight has up to 8 possible moves at each step.
2. Recursive Calls:
    - At each step, the algorithm explores all possible moves recursively.
    - The total number of moves to explore is approximately 8N28^{N^2}8N2, as there are N2N^2N2 squares on the board.
3. Backtracking Reduction:
    - Pruning invalid moves (those that are out of bounds or revisit squares) reduces the effective branching factor.
    - On average, the number of effective recursive calls is less than 8, but the worst-case complexity remains exponential.

Worst-case Time Complexity: O(8^{N^2})

This exponential growth makes the algorithm infeasible for large N.

Space Complexity:

The space complexity arises from:

1. Chessboard Representation:
    - The board is stored as a N×N matrix to track visited squares.
    - Space required: O(N^2).
2. Recursive Stack:
    - At most, N^2 recursive calls are made (one for each square).
    - Space required: O(N^2).

Overall Space Complexity: O(N^2)

9. CONCLUSION

The Knight's Tour problem is a classic example of using the **backtracking algorithm** to solve combinatorial problems. It demonstrates the systematic exploration of possible moves while adhering to constraints, such as staying within the chessboard and avoiding revisits.

Key takeaways include:

1. **Algorithmic Understanding**:
    - Backtracking provides a structured approach to solving problems with multiple constraints.
    - Optimization techniques like Warnsdorff's heuristic can reduce computation time in practical scenarios.
2. **Complexity Insights**:
    - The problem has exponential time complexity, making it computationally intensive for large chessboards.
    - Efficient space utilization ensures the algorithm is manageable for moderate values of NNN.
3. **Real-World Applications**:
    - Beyond chess, the Knight's Tour algorithm serves as a foundation for solving pathfinding problems in robotics, artificial intelligence, and game development.

By addressing the constraints and exploring all possible solutions, the Knight's Tour problem illustrates how computational challenges can be tackled effectively using algorithmic techniques.

10. REFERENCE

1. **Books**:
   - Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
     (For understanding backtracking algorithms and their applications.)
2. **Articles**:
   - Warnsdorff, H. C. (1823). *Knight's Tour Heuristic*.
     (For the heuristic approach in solving the Knight's Tour problem.)
3. **Web Resources**:
   - GeeksforGeeks. (n.d.). *Knight's Tour Problem – Backtracking*. Retrieved from https://www.geeksforgeeks.org.
     (Detailed explanation and pseudocode for the Knight's Tour problem.)
   - Programiz. (n.d.). *Backtracking Algorithm – Overview*. Retrieved from https://www.programiz.com.
     (Comprehensive guide to backtracking algorithms and their implementation.)