Knights tour problem

BY-
TINA KASHYAP- RA2211028010226
AKSHAY REDDY- RA2211028010229
KARTHIKEYA NALLA- RA2211029010015

CONTENTS

ABSTRACT

The Knight's Tour Problem is a classic chess puzzle that challenges the development of efficient algorithms to find a sequence of moves for a knight on a chessboard to visit each square exactly once. This project aims to design and analyze various algorithms to solve the Knight's Tour Problem, investigating their efficiency and performance.

The project will begin with a comprehensive review of existing algorithms for solving the Knight's Tour Problem, including backtracking, heuristic methods, and optimization techniques. The design phase will involve proposing novel algorithmic approaches and optimizations to enhance the efficiency of finding a solution

ABSTRACT

The analysis component of the project will focus on evaluating the proposed algorithms in terms of time complexity, space complexity, and solution optimality. Performance metrics such as the number of moves, computation time, and memory usage will be considered to compare the efficiency of the algorithms. Additionally, the project will explore the impact of varying chessboard sizes on algorithm performance.

The implementation phase will involve coding the proposed algorithms and conducting experiments on different chessboard sizes. The results will be analyzed to identify the strengths and weaknesses of each algorithm, allowing for a comprehensive understanding of their behavior under various scenarios.

Furthermore, the project will discuss the practical applications of the Knight's Tour Problem and how the developed algorithms can be extended to solve similar graph traversal problems. Insights gained from this project will contribute to the broader field of Design and Analysis of Algorithms (DAA) by providing a deeper understanding of algorithmic strategies for solving complex combinatorial problems.

PROBLEM SOLVING ALGORITHM

01 NAÏVE METHOD

02 BACKTRACKING ALGORITHM

03 WARNSNOFF'S ALGORITHM

04 DEPTH FIRST SEARCH (DFS)

PROBLEM SOLVING ALGORITHMS

1. Naive Method

The Naive method for solving the Knight's Tour problem involves trying out all possible sequences of knight moves starting from a given square and backtracking when a dead end is reached. While this method is conceptually simple, it is computationally expensive and often impractical for larger chessboards. The time complexity of the Naive method is O(n^(n^2)), making it unsuitable for boards larger than 8x8.

PROBLEM SOLVING ALGORITHMS

2. Backtracking Method

The Backtracking method is a more efficient approach for solving the Knight's Tour problem. It relies on the idea of exploring the chessboard incrementally and undoing moves when a solution cannot be found. Here are the key steps of the Backtracking method: • Start from an initial position on the chessboard. • Make a move to an unvisited square. • Repeat step 2 until all squares have been visited. • If no valid moves are possible from the current position, backtrack to the previous position and try a different move. • Continue backtracking until a solution is found or all possibilities are exhausted. The Backtracking method significantly reduces the number of possibilities that need to be explored compared to the Naive method, and it guarantees finding a solution if one exists.

3. Warnsdorff's Method

Warnsdorff's algorithm is a heuristic approach to solving the Knight's Tour Problem, a classic chess puzzle where the goal is to find a sequence of moves for a knight on a chessboard such that it visits each square exactly once. The algorithm is based on the idea of choosing the move that leads to a square with the fewest accessible unvisited neighbors.

The efficiency of Warnsdorff's algorithm lies in its ability to prioritize moves that lead to squares with fewer accessible unvisited neighbors. By doing so, the algorithm tends to explore the chessboard in a manner that increases the chances of finding a solution. However, it's important to note that Warnsdorff's algorithm does not guarantee a solution for all initial positions, and in some cases, it may fail to find a valid knight's tour.

PROBLEM SOLVING ALGORITHM

4. DEPTH FIRST SEARCH (DFS)

The search algorithm we will use to solve the knight's tour problem is called **depth first search** (**DFS**). Whereas the breadth first search algorithm discussed in the previous section builds a search tree one level at a time, a depth first search creates a search tree by exploring one branch of the tree as deeply as possible. In this section we will look at two algorithms that implement a depth first search. The first algorithm we will look at directly solves the knight's tour problem by explicitly forbidding a node to be visited more than once. The second implementation is more general, but allows nodes to be visited more than once as the tree is constructed. The second version is used in subsequent sections to develop additional graph algorithms

ALGORITHMS

NAIVE ALGORITHM

The Naive Algorithm is to generate all tours one by one and check if the generated tour satisfies the constraints.

```
while there are untried tours
{
  generate the next tour
  if this tour covers all squares
  {
     print this path;
  }
}
```

NAÏVE ALGORITHM

Naive Algorithm for Knight's Tour:

- Start with an empty solution vector.
- Choose an initial square for the knight on the chessboard.
- Mark the starting square as visited.
- For each valid move from the current square:

Make the move.Recursively repeat the process for the new square.
If the move leads to a solution (i.e., all squares are visited),
print the path.
Backtrack if necessary by removing the last move added to the solution vector.

BACKTRACKING ALGORITHM

- Start from an initial position on the chessboard.
- Make a move to an unvisited square. ·
- Repeat step 2 until all squares have been visited.
- If no valid moves are possible from the current position, backtrack to the previous position and try a different move.

BACKTRACKING ALGORITHM

- If all squares are visited
  print the solution
- Else

  a) Add one of the next moves to solution vector and recursively check if this move leads to a solution. (A Knight can make maximum eight moves. We choose one of the 8 moves in this step).

  b) If the move chosen in the above step doesn't lead to a solution then remove this move from the solution vector and try other alternative moves.

  c) If none of the alternatives work then return false (Returning false will remove the previously added item in recursion and if false is returned by the initial call of recursion then "no solution exists")

WARNSDOFF'S ALGORITHM

Algorithm:

1. Set P to be a random initial position on the board
2. Mark the board at P with the move number "1"
3. Do following for each move number from 2 to the number of squares on the board:

- let S be the set of positions accessible from P.
- Set P to be the position in S with minimum accessibility
- Mark the board at P with the current move number

4. Return the marked board – each square will be marked with the move number on which it is visited.

USING DEPTH FIRST SEARCH(DFS)

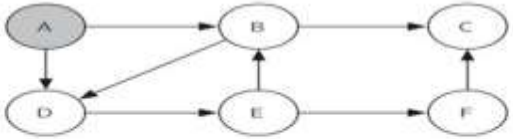

Figure 3: Start with node A

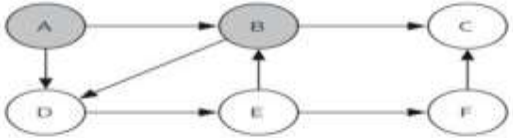

Figure 4: Explore B

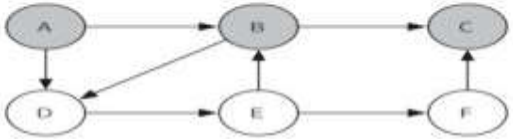

Figure 5: Node C is a dead end

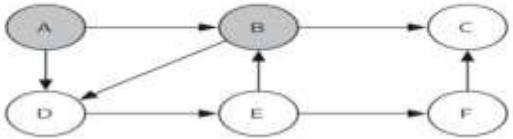

Figure 6: Backtrack to B

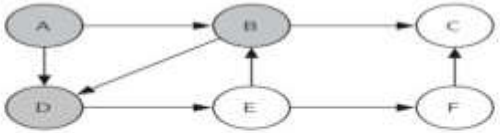

Figure 7: Explore D

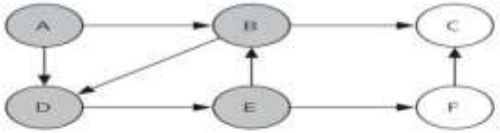

Figure 8: Explore E

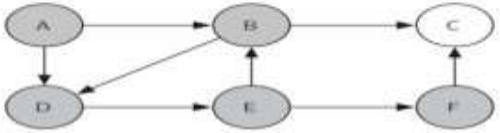

Figure 9: Explore F

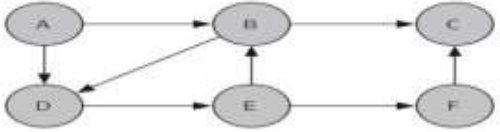

Figure 10: Finish

BACKTRACKING WITH DFS

1. **Initialize** the chessboard and the solution vector.
2. **Start** from the initial position (usually the top-left corner).
3. **Explore** all possible knight moves from the current position:

    1. Calculate the next cell coordinates based on knight's moves (e.g., moving two steps horizontally and one step vertically, or vice versa).
    2. Check if the move is within the board boundaries and hasn't been visited yet.
    3. If valid, mark the cell as visited and add it to the solution vector.
    4. Recursively explore further from the new position.



BACKTRACKING WITH DFS

4. Backtrack:

1. If the current path doesn’t lead to a solution, remove the last move from the solution vector and unmark the cell.
2. Try other alternatives.

5. Base Case:

1. If all squares are visited, **print the solution**.
2. Otherwise, continue exploring.

RECURSIVE FUNCTION OF THE ALGORITHM

```
#include <stdio.h>
#include <stdbool.h>

#define N 8

int xMoves[] = {2, 1, -1, -2, -2, -1, 1, 2};
int yMoves[] = {1, 2, 2, 1, -1, -2, -2, -1};

void printSolution(int sol[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf("%2d ", sol[i][j]);
        printf("\n");
    }
}
```

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

bool solveKTUtil(int x, int y, int movei, int sol[N][N]) {
    if (movei == N * N)
        return true;

    for (int k = 0; k < 8; k++) {
        int nextX = x + xMoves[k];
        int nextY = y + yMoves[k];
        if (isSafe(nextX, nextY, sol)) {
            sol[nextX][nextY] = movei;
            if (solveKTUtil(nextX, nextY, movei + 1, sol))
                return true;
            else
                sol[nextX][nextY] = -1; // backtrack
        }
    }

    return false;
}
```

```
bool solveKT() {
    int sol[N][N];

    // Initialize the solution matrix
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            sol[i][j] = -1;

    // Start from the top-left corner (0, 0)
    sol[0][0] = 0;

    if (!solveKTUtil(0, 0, 1, sol)) {
        printf("Solution does not exist");
        return false;
    } else {
        printf("Solution found:\n");
        printSolution(sol);
    }

    return true;
}

int main() {
    solveKT();
    return 0;
}
```

In this code, **solveKT** initializes the chessboard and starts the recursive solving process, while **solveKTUtil** is the recursive function that explores different moves. The **isSafe** function checks if a move is valid, and **printSolution** is used to print the final solution. The base case for the recursion is when all squares have been visited (**movei == N * N**), at which point the function returns **true**.
The recursive equation for the given code can be expressed as follows:

[T(N) = 8.[T(N-1)]

Here, (T(N)) represents the time complexity of the `solveKTUtil` function for a chessboard of size N X N . The function explores up to 8 possible moves at each step, leading to a recurrence relation where the time complexity at a given size depends on the time complexity at the previous size (N-1) multiplied by a constant factor of 8.

It's important to note that this recurrence relation is a simplification and assumes that each move takes constant time. In reality, the time complexity can be influenced by various factors such as the overhead of function calls, array accesses, and other operations within the function. Additionally, backtracking algorithms like this one often have a high degree of variability in their actual running time due to the nature of exploring different paths.

MASTER’S THEOREM

The Master Theorem is typically used to analyze the time complexity of divide-and-conquer algorithms. However, the Knight's tour problem does not fit the structure of a divide-and-conquer algorithm, so applying the Master Theorem directly is not appropriate.

The Knight's Tour Problem, particularly when solved using a backtracking approach, doesn't fit the typical divide-and-conquer structure that the Master Theorem addresses. Instead, the time complexity analysis for the Knight's Tour Problem often involves counting the number of possibilities explored during the backtracking process.

MASTER’S THEOREM

The time complexity of the Knight's Tour Problem is typically expressed in terms of the number of moves made or the number of recursive calls made during the backtracking exploration. It often has an exponential time complexity, as the number of possibilities grows exponentially with the size of the chessboard.

In summary, the Master Theorem is not directly applicable to the time complexity analysis of the Knight's Tour Problem, as it is not a divide-and-conquer algorithm with the specific recursive structure that the Master Theorem addresses. Time complexity analysis for the Knight's Tour Problem is usually done based on the nature of the backtracking algorithm and the number of possibilities explored.

TIME COMPLEXITY

The time complexity of the Knight's Tour Problem is generally exponential, specifically O(8^(N^2)) or O(2^(3.5 * N)), where N is the size of the chessboard.

The Knight's Tour Problem is typically solved using backtracking, exploring all possible moves and paths until a solution is found or all possibilities are exhausted. In the worst case, the number of possible sequences of moves that need to be explored grows exponentially with the size of the chessboard.

The branching factor in the search tree for the Knight's Tour Problem is 8 (since a knight has 8 possible moves at each position). With a chessboard of size N x N, there are N^2 positions to explore. As a result, the number of nodes in the search tree becomes 8^(N^2).

This exponential growth makes the Knight's Tour Problem computationally expensive for large chessboard sizes. Various heuristics and optimizations, such as Warnsdorff's rule to guide the search, can be applied to improve the efficiency, but the problem remains fundamentally challenging due to its combinatorial nature.

SPACE COMPLEXITY

- **Space Complexity Analysis :**
  - The primary contributor to space complexity is the recursive call stack.
  - The maximum depth of the call stack is influenced by the size of the chessboard (N x N) and the specific path taken during the exploration.
  - In the worst case, the maximum depth of the call stack can be N x N.
  - Therefore, the space complexity is O(N^2), where N is the size of the chessboard.
- It's important to note that the space complexity of backtracking algorithms can vary based on implementation details and optimizations. In the case of the Knight's Tour Problem, the recursive backtracking approach leads to a space complexity that grows with the size of the chessboard.

CONCLUSION

The Knight's Tour Problem remains a fascinating challenge in the realms of mathematics and computer science. While various algorithms have been developed to address it, the problem's inherent complexity continues to inspire researchers and enthusiasts to explore innovative solutions. The puzzle not only serves as an engaging recreational activity but also highlights the importance of algorithmic efficiency in solving real-world optimization problems.

REFERENCES

- The Knight's tour problem – GeeksforGeeks
- Warnsdorff's algorithm for Knight's tour problem – GeeksforGeeks
- Warnsdorff's algorithm for Knight's tour problem (includehelp.com)
- Knight's tour problem using backtracking and its analysis (codesdope.com)

THANKS!

CREDITS: This presentation template was created by **Slidesgo**, and includes icons by **Flaticon** and infographics & images by **Freepik**