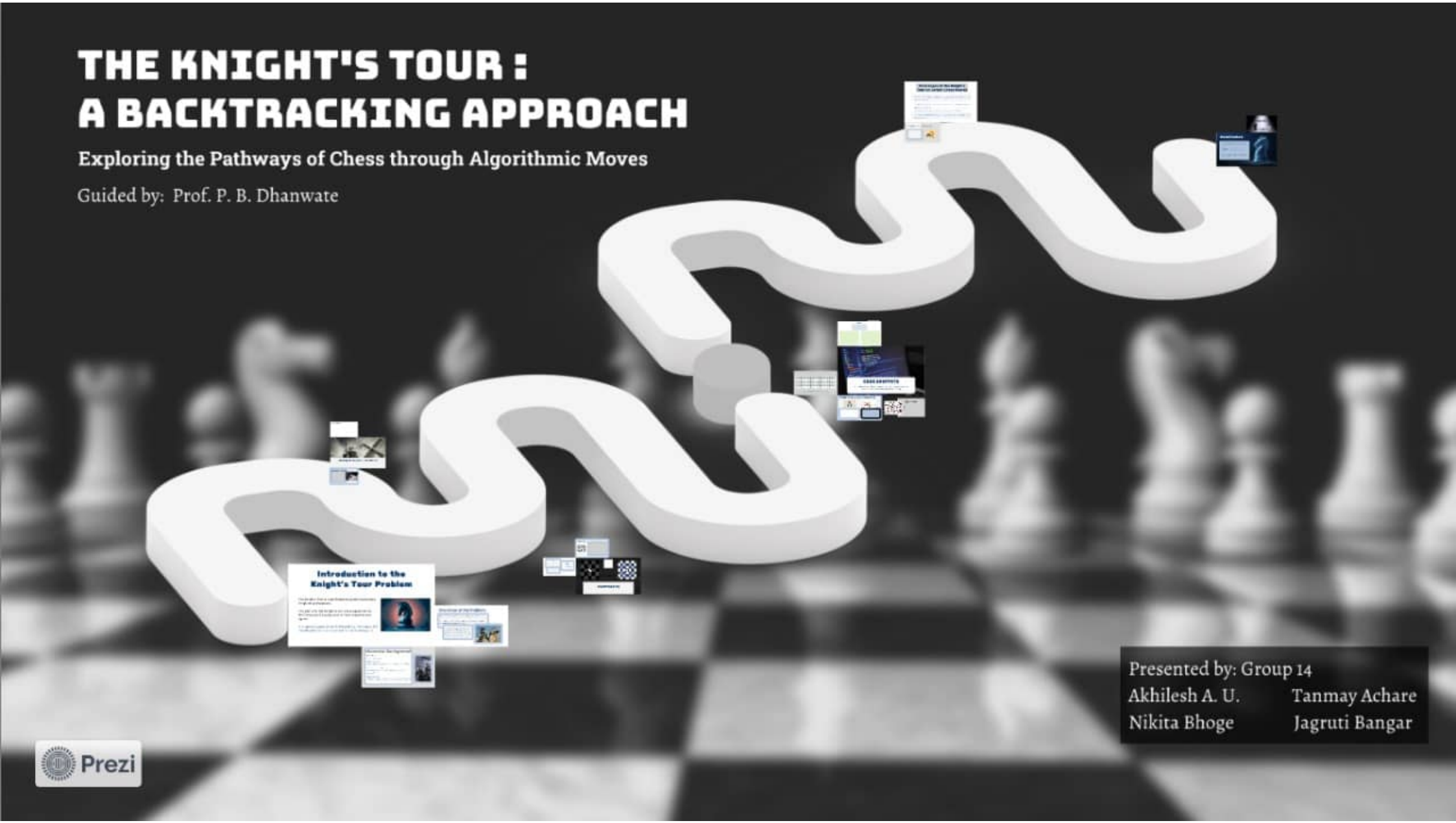
THE KNIGHT'S TOUR : A BACKTRACKING APPROACH

Exploring the Pathways of Chess through Algorithmic Moves

Guided by: Prof. P. B. Dhanwate



Presented by: Group 14
Akhilesh A. U. Tanmay Achare
Nikita Bhoge Jagruti Bangar

Prezi

Introduction to the Knight's Tour Problem

The Knight's Tour is a mathematical puzzle involving a knight on a chessboard.

The goal is for the knight to visit every square on the N×N chessboard exactly once without repeating any square.

It is typically posed on an N×N board (e.g., the classic 8x8 chessboard), but it can be solved on boards of any size.

Overview of the Problem

Objective: The Knight's Tour problem requires the knight to visit every square on an N×N chessboard exactly once.

The Knight's Tour is a classic example of a pathfinding problem and can be solved using backtracking.

Backtracking is particularly useful here as it allows exploring possible paths while discarding those that lead to dead ends.

Historical Background

- **Ancient Origins**
  The problem dates back to the 9th century in India and Persia, initially as a chess strategy exercise.

- **Medieval Popularity**
  Spread through Asia and Europe, becoming a common intellectual puzzle among scholars and nobility.

- **Euler's Contributions (18th Century)**
  Mathematician Leonhard Euler formally studied the Knight's Tour, introducing concepts of "closed tours" and laying the groundwork for graph theory.

- **Modern Significance**
  Now a classic example in computer science, demonstrating backtracking, recursion, and combinatorial optimization in AI, robotics, and pathfinding.

Defining the Knight's Tour Problem

Understanding the Knight's movement and the specific rules governing the Knight's Tour is essential for solving this intriguing chess problem.

Knight's Movement

Knight's Unique Movement:

- The knight moves in an "L" shape:
- Two squares in one direction (vertically or horizontally)
- and then one square perpendicular, or
- One square in one direction and then two squares perpendicular.

Key Characteristics

- **Jumping Ability:** Unlike other chess pieces, the knight can jump over other pieces, moving directly to its destination.
- **Eight Possible Moves:** From any position, a knight can typically make up to eight moves
- **Restricted Mobility:** The knight's movement is limited to specific squares, impacting its accessibility across the board.

Approaches:

Brute-Force Approach : Explore all possible knight moves until a valid path is found.

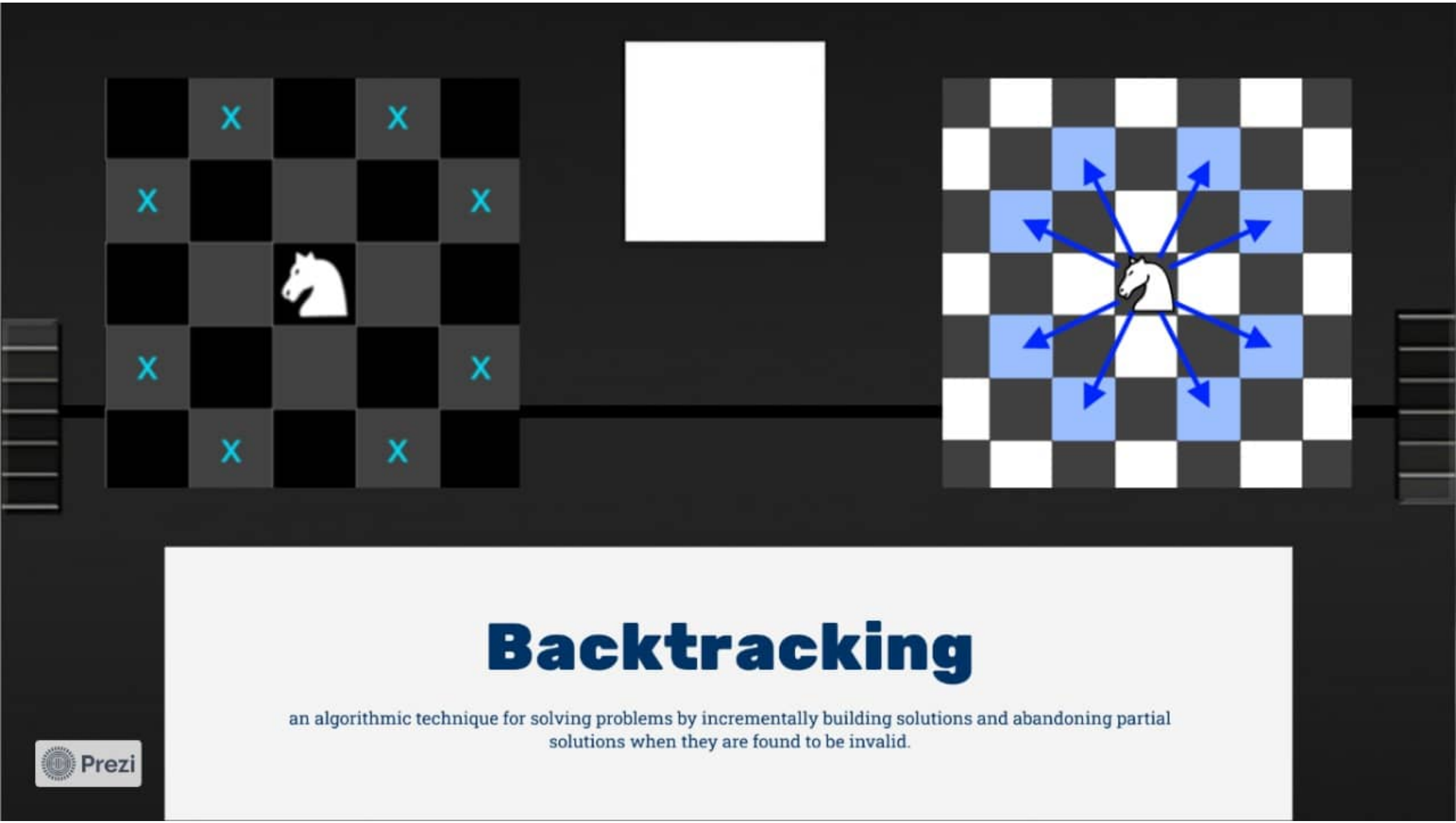
Backtracking Algorithm: Recursive method that explores paths and backtracks when a dead-end is reached.

Warnsdorff's Rule (Heuristic Approach): Prioritize moves to squares with fewer onward moves, improving path efficiency.

Backtracking with Warnsdorff's Rule: Combines backtracking with Warnsdorff's heuristic to optimize the solution.

Dynamic Programming (Alternative Method): Used in variations where overlapping subproblems can be stored and reused.

Graph Theory Approach: Framed as a Hamiltonian path problem where each square is a node, and each move is an edge.



Backtracking

an algorithmic technique for solving problems by incrementally building solutions and abandoning partial solutions when they are found to be invalid.

Backtracking

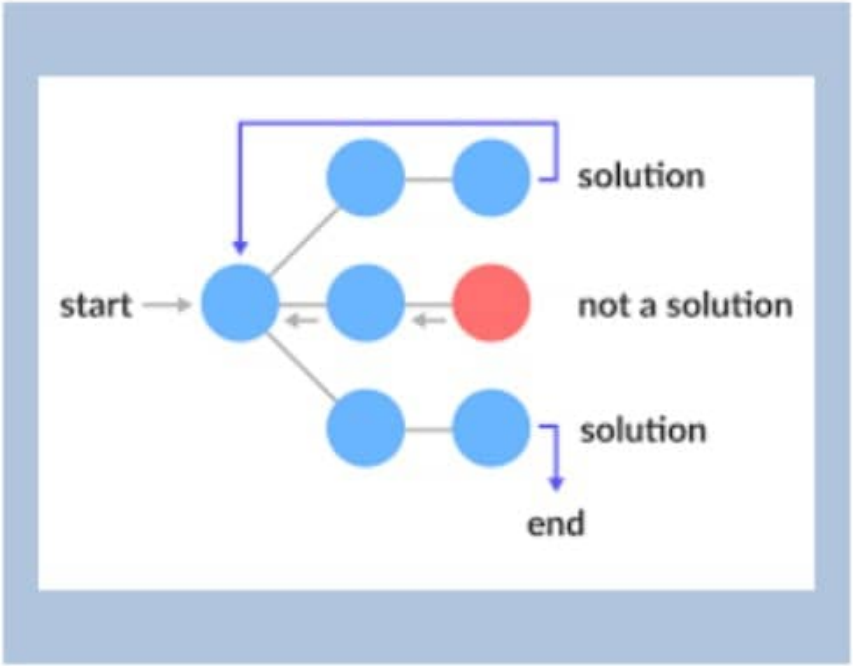
- Backtracking is a problem-solving algorithmic technique used to find solutions to problems by exploring all possible options and undoing choices when they lead to dead ends.
- It is a refinement of brute-force search that involves building solutions step by step and backtracking when a solution path is found to be invalid.

How Backtracking Works :

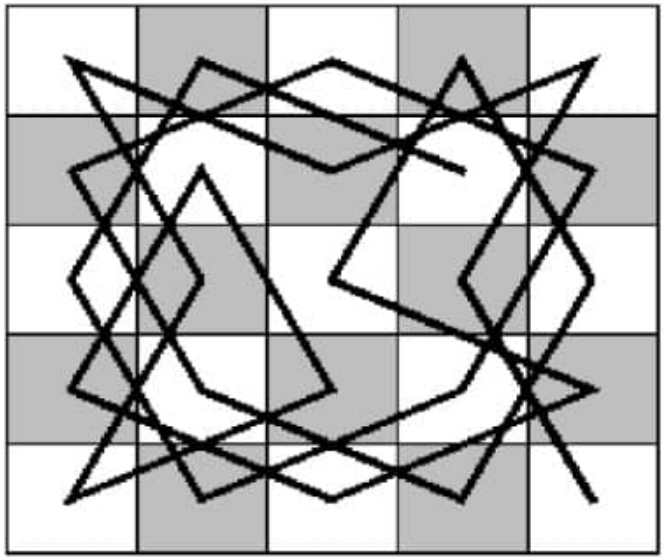
- Recursive Nature: Backtracking typically involves a recursive approach where each step tries to build upon the previous step. If the current path does not lead to a solution, the algorithm backtracks to the previous state and tries another path.
- Pruning Invalid Paths: The main idea is to stop pursuing certain branches of a decision tree when they are found to be unpromising or invalid, thereby saving computational time.

Backtracking in the Knight's Tour

- In the Knight's Tour problem, backtracking allows us to explore potential moves for the knight and abandon paths that do not lead to a solution (i.e., paths where the knight cannot visit all squares).



Algorithm



Check if All Squares are Visited:

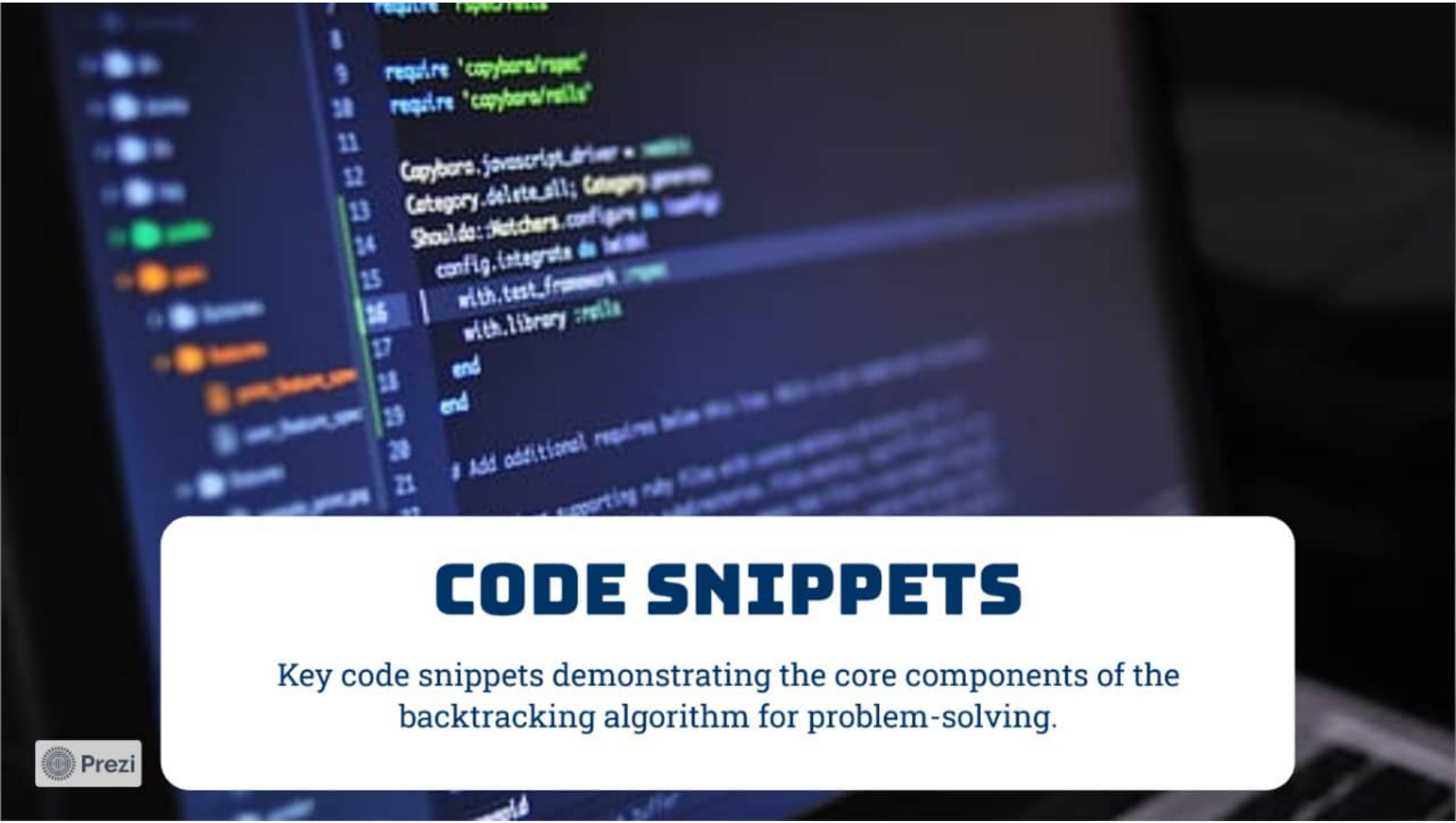
- If yes, print the solution.

Recursive Exploration:

- If not all squares are visited:
  - Try Next Move:
    - Add one of the next possible moves to the solution vector.
    - Recursively check if this move leads to a solution (choose one of the 8 possible knight moves).
  - Backtrack if Invalid:
    - If the current move doesn't lead to a solution, remove it from the solution vector and try another move.
  - Return False if No Move Works:
    - If no alternative moves are valid, return false (indicating no solution from the current path).

Base Case:

- If recursion returns false, backtrack and explore previous moves. If no solution exists, display "No solution exists".



CODE SNIPPETS

Key code snippets demonstrating the core components of the backtracking algorithm for problem-solving.

Code:

Checking Valid Moves:

```
def isValid(x, y, board):
    return 0 <= x < N and 0 <= y < N
and board[x][y] == -1
```

Initialization and Starting the Knight's Tour:

```
def solveKT():
    board = [[-1 for _ in range(N)] for _ in range(N)]
    xMove = [2, 1, -1, -2, -2, -1, 1, 2]
    yMove = [1, 2, 2, 1, -1, -2, -2, -1]
    board[0][0] = 0

    if not solveKTUtil(0, 0, 1, board, xMove, yMove):
        print("Solution does not exist")
    else:
        printSolution(board)
```

Recursive Backtracking Function:

```
def solveKTUtil(x, y, movei, board, xMove, yMove):
    if movei == N * N:
        return True
    for i in range(8):
        next_x = x + xMove[i]
        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
```

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

The path followed by Knight to cover all the cells

Performance and Limitations

Time Complexity :
There are N2 Cells and for each, we have a maximum of 8 possible moves to choose from, so the worst running time is O(8N^2).

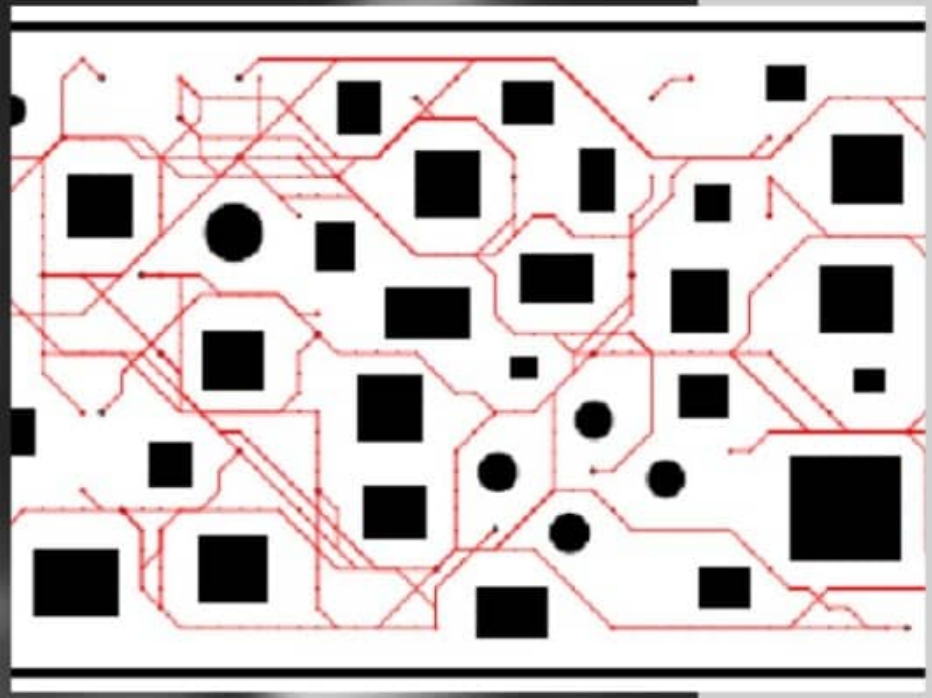
Auxiliary Space: O(N2)

Efficiency: Works well for smaller problem sizes or problems with many constraints that help prune invalid paths early.

High Computational Cost: The exponential time complexity makes backtracking impractical for large chessboards, leading to long processing times.

Inefficiency in Larger Searches: With no inherent optimization, backtracking explores many dead-end paths, resulting in redundant calculations.

Memory Usage: Recursive calls in backtracking can lead to high memory consumption, especially for larger values of N



Applications

- Testing and Benchmarking Algorithms
- AI and Game Development
- Robotic Path Planning
- Network Traversal and Graph Theory
- Computer Graphics and Image Processing
- Puzzle and Maze Design

Challenges of the Knight's Tour on Larger Chess Boards

Exponential Growth:

As the size of the chessboard increases (e.g., 16x16, 32x32), the number of possible paths grows exponentially. For an N×N board, there can be up to O(8^N^2) possible moves, making exhaustive search highly impractical.

Backtracking Limitations:

- Computational Time: The backtracking approach involves exploring all possible moves, which is manageable for smaller boards but becomes excessively time-consuming on larger boards due to the sheer number of recursive calls.
- Memory Usage: Recursive backtracking on larger boards consumes a significant amount of memory. Each call adds to the call stack, which can lead to stack overflow on very large boards.

Increased Dead-Ends:

Path Complexity: As board size grows, the knight is more likely to encounter paths that do not cover all squares, leading to increased dead-ends and repeated backtracking. This creates inefficient paths and longer search times.

Alternative Solutions:

Heuristics: Techniques like Warnsdorff's Rule can prioritize moves, reducing unnecessary backtracking.

Heuristic Search Algorithms: Algorithms such as genetic algorithms or simulated annealing can offer faster, though not guaranteed, solutions for larger boards.

Approximation Methods: For very large boards, approximate solutions may be more feasible than exhaustive backtracking, especially when perfect coverage isn't essential.

Conclusion

- The Knight's Tour problem using backtracking showcases an elegant but computationally expensive solution for small to moderate-sized boards (up to 8x8).
- While backtracking explores all possible moves, its exponential time complexity makes it impractical for larger boards.
- Optimizations like Warnsdorff's Rule can improve efficiency, but for very large boards, alternative methods such as heuristic algorithms may be required.
- Despite these challenges, the Knight's Tour remains a valuable problem in AI, robotics, and optimization, highlighting the importance of backtracking and the need for optimization in real-world applications.