**PROJECT**

**Report on**

**‘N’ QUEEN’S PROBLEM SOLVING**

**by**

Subhash Kumar (202410116100211)

Shubham Singh (202410116100204)

**Session:2024-2025 (II Semester)**

Under the supervision of

# Ms. Komal Salgotra Assistant Professor

KIET Group of Institutions, Delhi-NCR, Ghaziabad



DEPARTMENT OF COMPUTER APPLICATIONS

**KIET GROUP OF INSTITUTIONS, DELHI-NCR,**

**GHAZIABAD-201206**

( 2024- 2025)

# TABLE OF CONTENTS

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No** |  | **Section** | **Page Number** |
|  | Table of Contents |  |  |
| 1. | Introduction |  | 1 |
| 2. | Methodology |  | 2 |
| 3. | Code |  | 3-5 |
| 4. | Outcomes |  | 6 |

**CHAPTER - 01**

# INTRODUCTION

**1. Overview**

The **N-Queens Problem** is a classic combinatorial problem in which the goal is to place **N queens** on an **N × N chessboard** such that no two queens attack each other. This means that:

* No two queens share the same **row**.
* No two queens share the same **column**.
* No two queens share the same **diagonal** (both main and secondary).

The problem is a fundamental example of **backtracking algorithms**, demonstrating efficient searching and constraint satisfaction. It is widely used in artificial intelligence, algorithm optimization, and computational mathematics.

**2. Importance & Applications**

The N-Queens problem serves as an excellent case study in algorithmic thinking, particularly in:  
 **Backtracking Algorithms** – A recursive technique that explores possible solutions and backtracks when constraints are violated.  
 **Constraint Satisfaction Problems (CSPs)** – Common in AI, where multiple constraints need to be satisfied.  
 **Parallel Computing & Optimization** – Variants of this problem are used in scheduling and optimization tasks.  
 **Game Theory & Chess AI Development** – Understanding movement patterns and strategy-building.

**3. Objective of the Report**

This report aims to:

* Explain the **algorithmic approach** used to solve the N-Queens problem.
* Provide a **Python-based implementation** that solves and visualizes the problem.
* Discuss **time complexity, challenges, and optimizations** in solving the problem efficiently.

By the end of this report, readers will understand how to implement and visualize the N-Queens problem using Python and Matplotlib.

**CHAPTER - 02**

# METHODOLOGY

**1. Problem Formulation**  
The N-Queens problem requires placing N queens on an N × N chessboard such that no two queens threaten each other. This means that no two queens can share the same row, column, or diagonal. The problem is a classic example of a constraint satisfaction problem (CSP) and is typically solved using backtracking.

**2. Solution Approach**  
We use a backtracking algorithm to explore valid placements row by row while pruning invalid paths. The chessboard is represented as a 1D array, where board[i] = j indicates that a queen is placed in row i and column j. This reduces space complexity from O(N²) to O(N).

**3. Implementation Details**

* **Checking for Safe Placement:** Before placing a queen, we ensure no other queen exists in the same column or diagonal.
* **Recursive Backtracking:** The algorithm places queens one row at a time, backtracking when no valid placement exists.
* **Visualization:** A graphical representation is generated using Matplotlib, displaying queens with the Unicode symbol '♛' on a standard chessboard layout.

**4. Time Complexity Analysis**  
The worst-case time complexity is O(N!), as all permutations are explored in the absence of pruning. However, backtracking significantly reduces unnecessary computations.

**CHAPTER - 03**

# CODE

import numpy as np

import matplotlib.pyplot as plt

def draw\_chessboard(n, solution):

    """Draws a proper chessboard and places queens using '♛'"""

    fig, ax = plt.subplots(figsize=(n, n))

    # Draw the chessboard with alternating colors

    for i in range(n):

        for j in range(n):

            color = "cornsilk" if (i + j) % 2 == 0 else "saddlebrown"

            ax.add\_patch(plt.Rectangle((j, n-1-i), 1, 1, color=color))

    # Place queens using Unicode symbol '♛'

    for row, col in solution:

        ax.text(col + 0.5, n - 1 - row + 0.5, '♛', fontsize=40, ha='center', va='center', color="black")

    # Set board limits and remove axis labels

    ax.set\_xlim(0, n)

    ax.set\_ylim(0, n)

    ax.set\_xticks([])

    ax.set\_yticks([])

    ax.set\_xticklabels([])

    ax.set\_yticklabels([])

    ax.set\_frame\_on(False)

    plt.show()

def is\_safe(board, row, col, n):

    """Checks if a queen can be placed at board[row][col]"""

    for i in range(row):

        if board[i] == col or abs(board[i] - col) == abs(i - row):

            return False

    return True

def solve\_n\_queens(n, row=0, board=[], solutions=[]):

    """Backtracking solution to N-Queens"""

    if row == n:

        solutions.append(board[:])

        return

    for col in range(n):

        if is\_safe(board, row, col, n):

            board.append(col)

            solve\_n\_queens(n, row + 1, board, solutions)

            board.pop()

def visualize\_n\_queens(n):

    """Finds and visualizes the first valid N-Queens solution"""

    solutions = []

    solve\_n\_queens(n, board=[], solutions=solutions)

    if solutions:

        draw\_chessboard(n, [(r, c) for r, c in enumerate(solutions[0])])

    else:

        print("No solution found.")

# Run visualization for 8-Queens

visualize\_n\_queens(8)

**CHAPTER - 04**

# OUTCOMES

