****

A report on

**Solving the N-Queens Problem**

Submitted in partial fulfillment of the requirements for

the Master of Computer Application.

**Submitted By:**

Preksha Ruhela 202410116100148Nikhil 202410116100132

Priyakant Tyagi 202410116100151

MCA 1st year (Sec-C)

**Submitted To:**

Ms. Komal Salgotra

Department of Computer Application

KIET Group of Institutions  
Ghaziabad, Uttar Pradesh

**Introduction**

The N-Queens problem is a classic and fundamental challenge in the realm of computer science and mathematics. It involves placing N queens on an N x N chessboard in such a manner that no two queens threaten each other. Specifically, no two queens can occupy the same row, column, or diagonal. This problem is a generalized version of the original 8-Queens problem, where the goal is to arrange eight queens on an 8 x 8 chessboard under the same constraints.

The N-Queens problem is widely regarded as a quintessential example of combinatorial optimization and constraint satisfaction. It has been extensively used as a teaching model to understand fundamental concepts in backtracking algorithms, recursive problem-solving, and state space pruning. The importance of the N-Queens problem extends beyond academics, as it is closely related to real-world applications such as task scheduling, register allocation, and solving puzzles involving spatial arrangements.

Solving the N-Queens problem efficiently requires a systematic exploration of possible queen placements while ensuring that each move adheres to the safety conditions. This report presents an explanation of the backtracking approach used to solve the N-Queens problem. Additionally, this report discusses the algorithm's logic, its implementation in Python, and visualizations of the solution using libraries such as matplotlib.

Through this study, the report aims to provide insights into the core principles of recursive algorithms and demonstrates how backtracking can be utilized to solve constraint satisfaction problems effectively.

**Methodology**

To solve the N-Queens problem, this project follows a **backtracking algorithm**, which systematically searches for solutions by exploring possible queen placements one step at a time. The approach recursively builds the solution row by row (or column by column), backtracking whenever an invalid arrangement is detected.

**1. Problem Formulation**

The problem is formulated as a **constraint satisfaction problem (CSP)** where:

* **Variables**: The rows (or columns) where queens need to be placed.
* **Constraints**: No two queens should be placed on the same row, column, or diagonal.

**2. Algorithm Design**

The solution is designed using a **backtracking-based recursive approach**:

* Starting from the first column (or row), the algorithm tries placing a queen in every row (or column) one by one.
* Before placing a queen, it checks whether the position is **safe** using the isSafe() function, which verifies:
  + No other queen exists in the same row to the left.
  + No queen exists on the upper-left diagonal.
  + No queen exists on the lower-left diagonal.
* If a valid position is found, the queen is placed, and the algorithm proceeds to the next column.
* If no valid position exists, the algorithm **backtracks** by removing the queen from the last placed position and trying the next available row.

**3. Implementation Steps**

**Step 1: Initialize Board**

A 2D list (matrix) is initialized to represent the chessboard. Each cell is set to 0 (empty) or 1 (queen placed).

**Step 2: isSafe() Function**

This utility function ensures that a queen can be safely placed in the current position by checking row, upper diagonal, and lower diagonal on the left side.

**Step 3: solveNQUtil() Function**

This is the core recursive function. It attempts to place queens column by column and calls itself recursively to solve the subproblems.

**Step 4: Backtracking**

If placing a queen leads to a dead end (no safe positions in the subsequent columns), the algorithm backtracks by removing the last placed queen and tries the next possibility.

**Step 5: Visualization**

After successfully solving the problem, the board configuration is visualized using **matplotlib**, where:

* Black and white squares mimic a real chessboard.
* A queen ('Q') is marked on the appropriate cells.

**4. Tools and Libraries Used**

* **Python**: The primary programming language for the implementation.
* **Matplotlib**: For visualizing the chessboard and the placement of queens.
* **Google Colab**: To execute the code in a cloud-based environment easily.

**5. User Input Integration**

The algorithm is designed to accept user input for **N**, allowing the solution of the N-Queens problem for any board size dynamically.

**6. Performance Consideration**

While the backtracking approach works efficiently for smaller values of **N** (e.g., N ≤ 20), the time complexity grows exponentially with larger values of **N** due to the combinatorial nature of the problem. Optimizations like forward-checking or heuristic methods (e.g., Least Constraining Value heuristic) can be used to improve efficiency but are outside the scope of this report.

**Code**

import matplotlib.pyplot as plt

import numpy as np

def visualizeBoard(board, N):

    fig, ax = plt.subplots()

**# Create the checkerboard pattern**

    for i in range(N):

        for j in range(N):

            color = 'white' if (i + j) % 2 == 0 else 'black'

            rect = plt.Rectangle((j, i), 1, 1, facecolor=color)

            ax.add\_patch(rect)

            if board[i][j] == 1:

                ax.text(j + 0.5, i + 0.5, 'Q', va='center', ha='center', fontsize=20, color='darkgreen', fontweight='bold')

    ax.set\_xlim(0, N)

    ax.set\_ylim(0, N)

    ax.set\_aspect('equal')

    ax.axis('off')

    plt.gca().invert\_yaxis()

    plt.title(f"{N}-Queens Solution", fontsize=16)

    plt.show()

def isSafe(board, row, col, N):

    for i in range(col):

        if board[row][i] == 1:

            return False

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

        if board[i][j] == 1:

            return False

    for i, j in zip(range(row, N, 1), range(col, -1, -1)):

        if board[i][j] == 1:

            return False

    return True

def solveNQUtil(board, col, N):

    if col >= N:

        return True

    for i in range(N):

        if isSafe(board, i, col, N):

            board[i][col] = 1

            if solveNQUtil(board, col + 1, N) == True:

                return True

            board[i][col] = 0

    return False

def solveNQ(N):

    board = [[0 for \_ in range(N)] for \_ in range(N)]

    if solveNQUtil(board, 0, N) == False:

        print("No solution exists for N = 2 or N = 3.")

    else:

        visualizeBoard(board, N)

**# Get input from user**

try:

    N = int(input("Enter the number of queens (N): "))

    if N < 1:

        print("Please enter a positive integer greater than 0.")

    else:

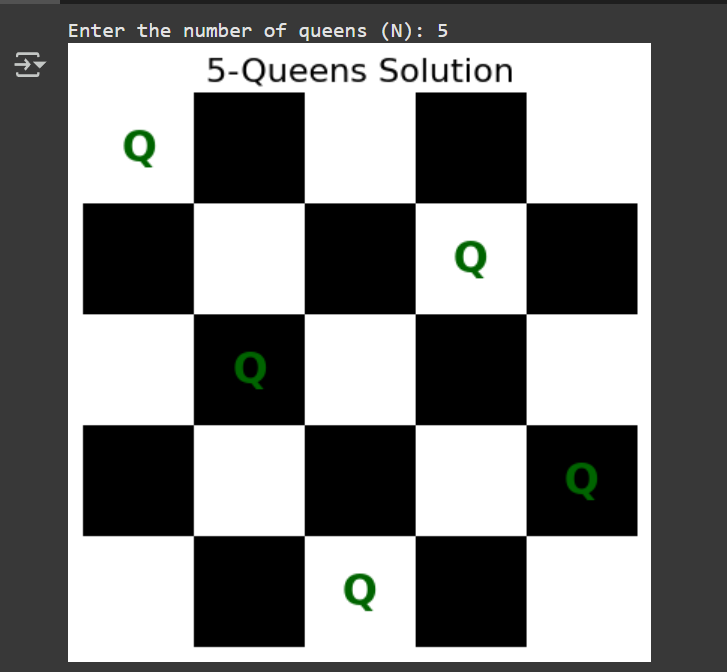
        solveNQ(N)

except ValueError:

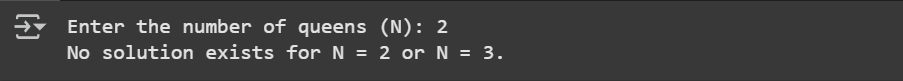
    print("Please enter a valid integer.")

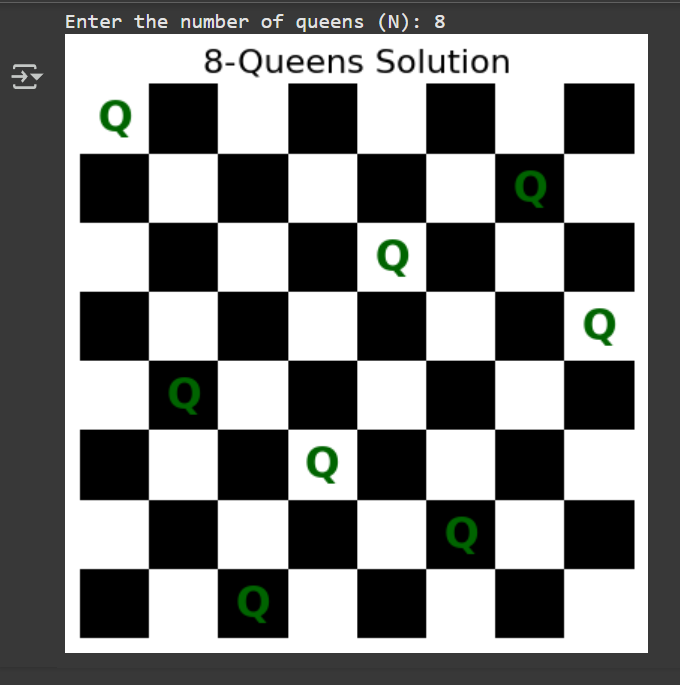
**Output**



****

****

****

****