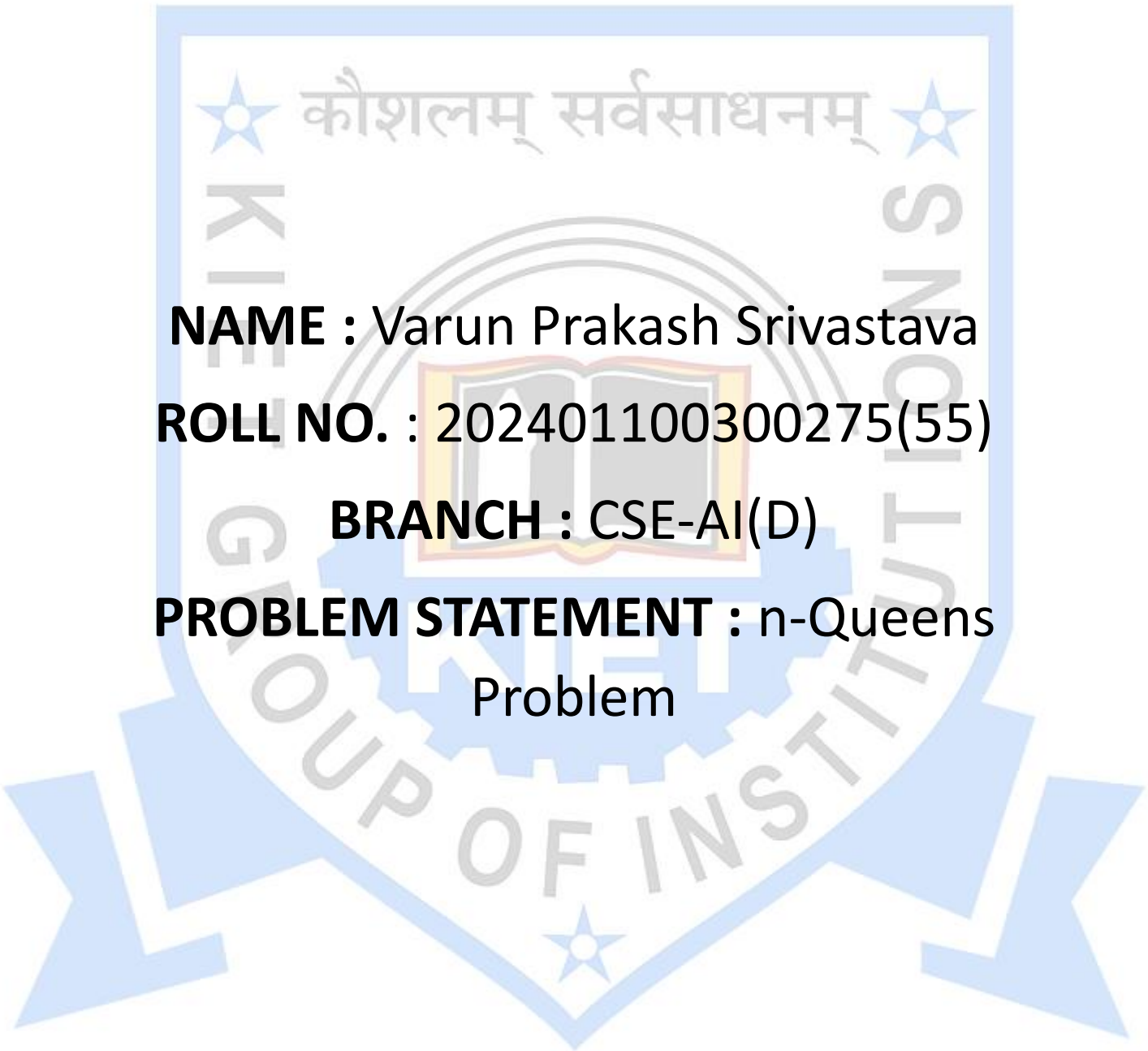


NAME : Varun Prakash Srivastava

ROLL NO. : 2024011003002 75(55)

BRANCH : CSE-AI(D)

PROBLEM STATEMENT : n-Queens Problem

Q. What is n-Queens Problem?

Ans. N-Queens is an important and traditional problem of computer science and artificial intelligence, more precisely combinatorial optimization and constraint satisfaction. The task is to put N queens on an N×N chessboard so that no two queens can attack each other according to the chess queen's move rules. A queen can travel any number of squares in any horizontal, vertical, or diagonal direction, i.e., potentially attack any other queen that lies in the same row, column, or diagonal. The problem is interesting and challenging because it is a question of determining a configuration where such attack possibilities do not exist. It poses problems to a variety of algorithmic methods, including backtracking, constraint satisfaction, and heuristics. Furthermore, as N increases, the problem becomes more difficult, making it therefore a perfect problem to use in investigating search and optimization techniques in AI and computer science.

Q. Which approach used to solve the problem?

Ans. The approach used in solving the N-Queens problem in the code is Hill Climbing with Random Restarts. Let me explain this method in simple terms.

Hill Climbing:

1. What is Hill Climbing?
    - Hill Climbing is a type of algorithm that tries to find a solution by starting with a random configuration (in this case, placing queens randomly on the chessboard).
    - The algorithm then tries to improve this configuration step by step by making small changes (like moving one queen to a different column in the same row).
    - After each change, the algorithm checks how many "attacks" (conflicts between queens) are present. The goal is to reduce the number of attacks.
    - The algorithm keeps improving the board until it finds a solution with zero attacks, meaning no queens are attacking each other.

2. How does it work in this code?
    - The algorithm starts by generating a random board where each queen is placed in a random column of each row.
    - It calculates how many queens are attacking each other by checking if they share the same diagonal.
    - The algorithm then tries to improve the board by moving each queen to a different column (one at a time) and sees if the new configuration results in fewer attacks.
    - If it finds a board with fewer attacks, it keeps that new configuration.
    - It repeats this process, moving queens around, until either a solution is found or no better configuration can be found. If no improvement is found, it restarts with a new random board.

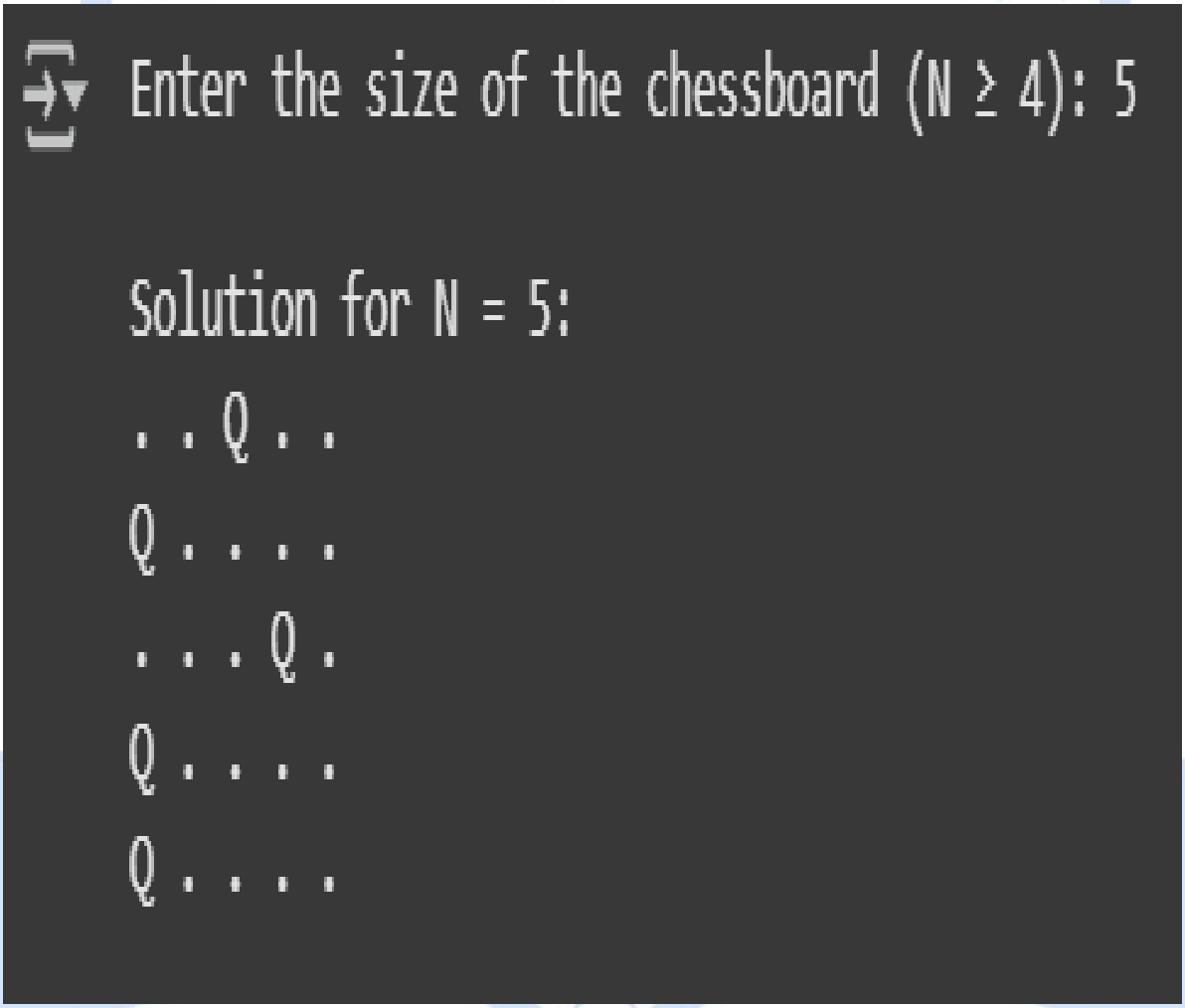
CODE

```python
import numpy as np
import random

def initialize_board(N):
    return np.random.permutation(N)

def compute_attacks(board):
    N = len(board)
    attacks = 0
    for i in range(N):
        for j in range(i + 1, N):
            # Check if two queens are on the same diagonal
            if abs(board[i] - board[j]) == abs(i - j):
                attacks += 1  # Increment attack count if they are attacking each other
    return attacks

def get_best_neighbor(board):
    N = len(board)
    best_board = board.copy()
    min_attacks = compute_attacks(board)  # Current attack count
```

```
    # Try moving each queen to a different column in the same row
    for row in range(N):
        for col in range(N):
            if board[row] == col:
                continue  # Skip if the queen is already in this column

            new_board = board.copy()  # Copy current board
            new_board[row] = col  # Move queen to a new column
            new_attacks = compute_attacks(new_board)  # Count new conflicts

            # Update best board if the move reduces conflicts
            if new_attacks < min_attacks:
                best_board = new_board
                min_attacks = new_attacks

    return best_board, min_attacks

def hill_climbing(N, max_restarts=100):
    for _ in range(max_restarts):  # Attempt multiple restarts if needed
        board = initialize_board(N)  # Start with a random board
        while True:
            current_attacks = compute_attacks(board)
            if current_attacks == 0:
                return board  # Found a solution with zero conflicts

            new_board, new_attacks = get_best_neighbor(board)

            if new_attacks >= current_attacks:
                break  # If no improvement, restart with a new board

            board = new_board  # Move to better configuration

    return None  # No solution found after max_restarts attempts
```

```python
def print_board(board):
    N = len(board)
    for row in range(N):
        line = ["Q" if board[row] == col else "." for col in range(N)]
        print(" ".join(line))  # Print row with queens and empty spaces
    print("\n")  # Add spacing between different solutions

# Take user input for board size
N = int(input("Enter the size of the chessboard (N ≥ 4): "))

# Validate input: N must be at least 4
while N < 4:
    print("N must be at least 4.")
    N = int(input("Enter a valid N (N ≥ 4): "))

# Solve the N-Queens problem
solution = hill_climbing(N)

# Display result
if solution is not None:
    print(f"\nSolution for N = {N}:")
    print_board(solution)
else:
    print("\nNo solution found.")
```

OUTPUT



CREDITS: ChatGPT