N-Queens Problem Report

Personal Details

Name: Shivani Yadav
Course : B.Tech
Branch: CSE (Artificial Intelligence) - D
Roll No.: 202401100300235

Introduction

The N-Queens problem is a famous challenge in mathematics and computer science. It was first introduced in the 19th century by Max Bezzel and later generalized. The problem has been widely studied due to its applications in optimization, artificial intelligence, and problem-solving techniques. The main challenge is to find a way to place N queens on an N × N chessboard while ensuring that none of them can attack each other. Since a queen can move any number of squares in a straight line (horizontally, vertically, or diagonally), placing them in a non-attacking manner requires careful planning.

This problem has been used in computational studies to evaluate different algorithmic strategies, including brute force, constraint satisfaction problems (CSP), and heuristics. One of the most efficient methods for solving this problem is using a backtracking algorithm, which systematically explores different placements while eliminating incorrect ones. The problem becomes increasingly difficult as N grows, requiring optimized solutions for larger values.

The N-Queens problem also has real-world applications, such as in parallel computing, circuit placement, and scheduling problems. It demonstrates how algorithms can be used to solve constraint-based optimization problems in different fields. By implementing an algorithmic approach, we can efficiently explore possible solutions and analyze their effectiveness.

Methodology

To solve the N-Queens problem, we use the **backtracking algorithm**, a powerful method for constraint satisfaction problems. Backtracking is a recursive approach where we try placing queens one by one and check if the placement is valid. If a conflict arises, we remove the last placed queen (backtrack) and try a different position. This method ensures that only valid configurations are explored, reducing unnecessary computations.

The approach follows these steps:

1. **Representing the Board:**
    - The chessboard is represented as an **N × N grid**, where each cell is either empty (False) or contains a queen (True).
2. **Checking Safety:**
    - Before placing a queen, we check if the position is **safe** by ensuring that no other queen is in the same **column**, **left diagonal**, or **right diagonal**.
    - We iterate through the rows above the current row to verify that placing a queen will not cause a conflict.
3. **Using Backtracking:**
    - We start placing queens **row by row**, ensuring each queen is in a valid position.
    - If a conflict is detected, we remove the last placed queen and move to the next possible position in the same row.
    - If a valid position is found, we move to the next row and repeat the process.
    - If no valid placement exists in a row, we backtrack to the previous row and adjust the queen's position.
4. **Shuffling Columns:**
    - To **introduce variety** in the solutions and avoid getting stuck in repetitive placements, we shuffle the column order before trying placements.
    - This ensures that different board configurations are explored.
5. **Taking User Input:**
    - The program allows the user to input different values of N.
    - The solver attempts to find a valid arrangement of queens for the given N.
    - If no solution exists for a particular N, the program informs the user accordingly.

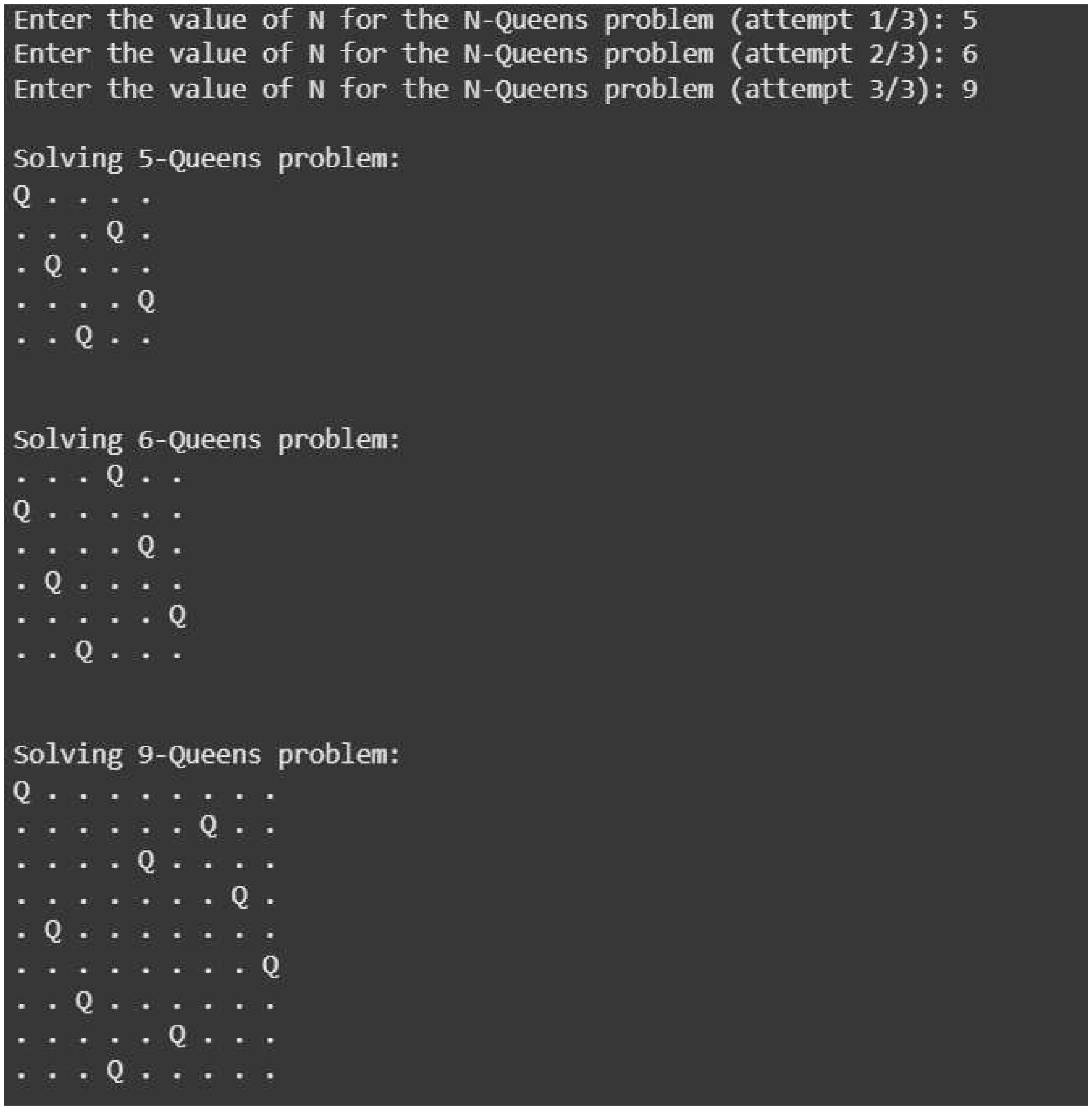
This methodology ensures an **efficient search** for solutions while minimizing unnecessary computations. Backtracking is a **systematic approach** that guarantees a solution if one exists, making it an effective strategy for solving the N-Queens problem.

Code

```
import random


def print_solution(board):
    """Prints the N-Queens board solution in a readable format."""
    for row in board:
        print(" ".join("Q" if col else "." for col in row))  # Print
row with 'Q' for queen, '.' for empty space
    print()  # Print a newline for readability


def is_safe(board, row, col, n):
    """Checks if it's safe to place a queen at board[row][col]."""
    # Check column for another queen
    for i in range(row):
        if board[i][col]:
            return False

    # Check upper-left diagonal for another queen
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j]:
            return False

    # Check upper-right diagonal for another queen
    for i, j in zip(range(row, -1, -1), range(col, n)):
```

```python
        if board[i][j]:
            return False

    return True  # No conflicts, safe to place queen

def solve_n_queens(board, row, n):
    """Uses backtracking to find a single valid solution for the N-Queens problem."""
    if row == n:  # Base case: all queens are placed successfully
        print_solution(board)  # Print the solution board
        return True  # Stop searching once a solution is found

    columns = list(range(n))
    random.shuffle(columns)  # Randomize column order for diversity in solutions

    for col in columns:  # Try placing queen in each column of the current row
        if is_safe(board, row, col, n):  # Check if it's safe to place queen
            board[row][col] = True  # Place queen in the board[row][col]
            if solve_n_queens(board, row + 1, n):  # Recur to place next queen
                return True  # Stop searching if a valid solution is found
            board[row][col] = False  # Backtrack: Remove queen and try next position
    return False  # No valid solution found in this path


def n_queens(n):
    """Initializes the board and starts the N-Queens solver."""
    board = [[False] * n for _ in range(n)]  # Create an N x N board initialized with False
    if not solve_n_queens(board, 0, n):  # Start solving from the first row
        print("No solution exists")  # If no solution, print message


# Take user input for three board sizes
n_values = [int(input(f"Enter the value of N for the N-Queens problem (attempt {i+1}/3): ")) for i in range(3)]


for n in n_values:
    if n < 1:
        print(f"Invalid input! N must be at least 1 (N={n}).")  # Ensure valid input
    else:
        print(f"\nSolving {n}-Queens problem:")
        n_queens(n)  # Call the solver with the chosen N
```

Output



```
Enter the value of N for the N-Queens problem (attempt 1/3): 5
Enter the value of N for the N-Queens problem (attempt 2/3): 6
Enter the value of N for the N-Queens problem (attempt 3/3): 9

Solving 5-Queens problem:
Q . . . .
. . . Q .
. Q . . .
. . . . Q
. . Q . .


Solving 6-Queens problem:
. . . Q . .
Q . . . . .
. . . . Q .
. Q . . . .
. . . . . Q
. . Q . . .


Solving 9-Queens problem:
Q . . . . . . . .
. . . . . . Q . .
. . . . Q . . . .
. . . . . . . Q .
. Q . . . . . . .
. . . . . . . . Q
. . Q . . . . . .
. . . . . Q . . .
. . . Q . . . . .
```

Reference

1. "Artificial Intelligence: A Modern Approach" by Stuart Russell and Peter Norvig.
2. Online tutorials and learning resources on backtracking algorithms.
3. Classic approaches to solving the N-Queens problem used in computer science education.

This report explains the N-Queens problem in a simple way and shows how to solve it using Python.