Name:

Muhammad Awais

Section:

BSAI-4C-114

Subject:

PAI-Lab

Submitted to:

Sir Rasikh

Task Report

N-Queens Problem :

1. Introduction

The **N-Queens problem** is a combinatorial puzzle where `N` queens must be placed on an `N × N` chessboard such that no two queens threaten each other. This report details the implementation of the problem using **backtracking** to find all valid solutions.

2. Problem Definition

- **Board Size:** `N × N`
- **Constraints:**
  - No two queens share the same row.
  - No two queens share the same column.
  - No two queens are on the same diagonal.

3. Implementation Details

The problem is solved using a recursive **backtracking** approach.

3.1 Class Initialization

```
class NQueens:
    def __init__(self, n):
        self.n = n
        self.board = [-1] * n  # Stores column
placement for each row
        self.solutions = []  # Stores valid solutions
```

- `n`: Board size.
- `board`: List where `board[row] = col` indicates the queen’s position.
- `solutions`: Stores all valid configurations.

3.2 Main Solving Function

```
def solve(self):
```

```
    self.solve_util(0)
    return self.solutions
```

- Calls `solve_util(0)` to start placing queens from row 0.

3.3 Recursive Backtracking Function

```
def solve_util(self, row):
    if row == self.n:
        self.add_solution()
        return
    for col in range(self.n):
        if self.is_safe(row, col):
            self.board[row] = col
            self.solve_util(row + 1)
            self.board[row] = -1  # Backtrack
```

- **Base Case:** If all queens are placed, the solution is stored.
- **Column-wise Placement:** Checks each column for safe placement.
- **Recursive Call:** Moves to the next row if placement is valid.
- **Backtracking:** Removes the last placed queen to explore other configurations.

3.4 Checking Safe Placement

```
def is_safe(self, row, col):
    for i in range(row):
        if self.board[i] == col or abs(self.board[i] -
col) == abs(i - row):
            return False
    return True
```

- Ensures no two queens share the same **column** or **diagonal**.

3.5 Storing and Printing Solutions

```
def add_solution(self):
    solution = []
    for row in range(self.n):
        row_str = ['.'] * self.n
        row_str[self.board[row]] = 'Q'
```

```
        solution.append(''.join(row_str))
    self.solutions.append(solution)
```

- Converts the `board` list into a chessboard representation.
- Uses `Q` for queens and `.` for empty spaces.

```
def print_solutions(self):
    if not self.solutions:
        print("No solution exists.")
    else:
        for idx, solution in enumerate(self.solutions):
            print(f"Solution {idx + 1}:")
            for row in solution:
                print(row)
            print()
```

- Prints each solution in a readable chessboard format.

4. Example Execution

Input:

```
n = 4
nqueens = NQueens(n)
nqueens.solve()
nqueens.print_solutions()
```

Output (Example Solutions for N=4):

```
Solution 1:
.Q..
...Q
Q...
..Q.

Solution 2:
..Q.
Q...
...Q
.Q..
```

- Each solution represents a valid arrangement of queens.

5. Conclusion

- **Backtracking efficiently explores all possible placements**.
- **Uses pruning (`is_safe()`) to eliminate invalid placements early**.
- **Finds and prints all valid solutions for a given ****`N`**.

Screenshot:

```
Solution 1:
Q.......
....Q...
.......Q
.....Q..
..Q.....
......Q.
.Q......
...Q....

Solution 2:
Q.......
.....Q..
.......Q
..Q.....
......Q.
...Q....
.Q......
....Q...

Solution 3:
Q.......
......Q.
...Q....
.....Q..
...
.Q......
......Q.
....Q...
```