SUPERIOR UNIVERSITY

Name : Huzaifa Rehan

Roll No : SU92-BSAIM-F23-071

Section : AI(4-B)

Lab Task : 04

Submitted to : Sir Rasikh Ali

Lab Task -4 (N-Queen Problem)

N-Queen Problem Report

Introduction

The N-Queen problem is a classic combinatorial problem in computer science and artificial intelligence. It involves placing N queens on an N×N chessboard in such a way that no two queens threaten each other. This means that no two queens should share the same row, column, or diagonal. The problem is a generalization of the well-known 8-Queen problem.

Problem Definition

Given an N×N chessboard, the objective is to place N queens on the board so that:

1. No two queens share the same row.
2. No two queens share the same column.
3. No two queens share the same diagonal (both major and minor diagonals).

Applications of the N-Queen Problem

The N-Queen problem has several practical applications, including:

- **Constraint Satisfaction Problems (CSPs):** Used in artificial intelligence and operations research.
- **Parallel Processing:** Helps in optimizing parallel computing algorithms.
- **VLSI Design:** Used in designing circuits without conflicts.
- **Robotics:** Applied in autonomous decision-making processes.

Solution Approaches

Several methods exist to solve the N-Queen problem efficiently:

1. Backtracking Algorithm

Backtracking is a brute-force recursive approach that places queens one by one in different rows and backtracks when a conflict is encountered.

- It systematically explores possible solutions by placing a queen in a row and recursively solving the subproblem for the next row.
- If a solution is found, it is stored; otherwise, it backtracks and tries a different placement.
- Time Complexity: O(N!).

2. Branch and Bound

This optimization of backtracking reduces the number of possibilities by using bounding functions to eliminate infeasible solutions early.

- It improves efficiency by pruning the search tree.
- Time Complexity: Better than O(N!) but still exponential in nature.

3. Constraint Programming (CSP Approach)

- The problem is modeled as a constraint satisfaction problem.
- Uses constraint propagation techniques to reduce search space.
- Efficient for larger values of N.

4. Genetic Algorithms

- Uses evolutionary techniques to generate and improve solutions over generations.
- Suitable for approximate solutions rather than exact ones.
- Time Complexity varies depending on implementation.

5. Heuristic and Metaheuristic Approaches

- **Hill Climbing:** Places queens randomly and iteratively moves them to minimize conflicts.
- **Simulated Annealing:** Uses probabilistic techniques to escape local optima.

Complexity Analysis

The time complexity of solving the N-Queen problem varies based on the approach used:

- **Backtracking:** O(N!) (worst-case exponential complexity).
- **Branch and Bound:** More optimized than backtracking but still exponential.
- **Heuristic Methods:** Can achieve near-optimal solutions in polynomial time for larger N.

Challenges in the N-Queen Problem

1. **Exponential Growth:** The number of possible board configurations grows exponentially with N.
2. **Large N values:** Finding an exact solution for very large values of N (e.g., N > 1000) is computationally expensive.

3. **Memory Constraints:** Recursive methods may lead to excessive memory usage for large N.

Conclusion

The N-Queen problem is a fundamental problem in computer science with significant applications in artificial intelligence and optimization. While brute-force methods provide solutions for small values of N, heuristic and metaheuristic methods are necessary for solving large instances efficiently. The problem continues to be a rich area of research in algorithm design and constraint solving.

Code:

```python
class nqueen:
    def __init__(self,n):
        self.n = n
        self.board = [-1]*n
        self.solution = []

    def safe(self,row,col):
        for pre_row in range(row):
            pre_col = self.board[pre_row]
            if pre_col == col or abs(pre_col-col)==abs(pre_row-row):
                return False
        return True

    def solve_queen(self,row=0):
        if row==self.n:
            self.solution.append(self.board[:])
            return
        for col in range(self.n):
            if self.safe(row,col):
                self.board[row] = col
                self.solve_queen(row+1)
                self.board[row] = -1

```

```python

    def print_sol(self):
        for sol in self.solution:
            for row in range(self.n):
                l = ["." for _ in range(self.n)]
                l[sol[row]] = 'Q'
                print(" ".join(l))
            print('\n'+'-'*(2*self.n-1))
    def solve(self):
        self.solve_queen()
        print(f'total solution for {self.n}-Queen:{len(self.solution)}')
        self.print_sol()
if __name__ =='__main__':
    n = 8
    game = nqueen(n)
    game.solve()
```

```
total solution for 8-Queen:92
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

---------------
Q . . . . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
. . . . . . Q .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .
```