SUPERIOR UNIVERSITY

Programming for Artificial Intelligence – Lab

Task 4

Name: Minhaj Asghar

Roll no: SU92-BSAIM-F23-108

Section: BSAI-4B

Submitted to: Sir Rasikh Ali

N-Queens Problem - Report

Introduction

The N-Queens problem is a well-known combinatorial problem in computer science and artificial intelligence. The goal is to place N queens on an N x N chessboard such that no two queens can attack each other. A queen can attack another queen if they share the same row, column, or diagonal. The challenge is to find all possible solutions to this problem.

Problem Statement

Given a chessboard of size N x N, the task is to place N queens on the board such that no two queens threaten each other. The problem must be solved by:

- Ensuring that no two queens are in the same row, column, or diagonal.
- Finding all possible ways to place the queens.

Approach

The problem is solved using **backtracking**. The idea is to place queens one by one in each row, checking for conflicts in columns and diagonals. If a queen can be safely placed, we move on to the next row and repeat the process. If at any point no queen can be placed in a row, we backtrack, i.e., remove the last placed queen and try another position for it.

Steps:

1. **Place queens row by row**: Start with the first row, place a queen in one of the columns, and proceed to the next row.
2. **Check for safety**: Before placing a queen, check if it's safe by ensuring no other queens are in the same column or diagonals.
3. **Backtrack**: If placing a queen leads to a conflict in the next row, remove the queen and try the next column.
4. **Solution found**: Once all queens are placed safely, store the solution.

Code Implementation:

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

def solve_nqueens(board, row, n, result):
    if row == n:
        result.append(board[:])
        return

    for col in range(n):
        if is_safe(board, row, col, n):
            board[row] = col
            solve_nqueens(board, row + 1, n, result)
            board[row] = -1

def nqueens(n):
    board = [-1] * n
    result = []
    solve_nqueens(board, 0, n, result)

    for solution in result:
        for row in solution:
            line = ['Q' if col == row else '.' for col in range(n)]
            print(' '.join(line))
        print("\n")

n = 4
nqueens(n)
```

Explanation of Code:

1. **is_safe(board, row, col, n)**: This function checks whether it's safe to place a queen at the position (row, col). It checks if there's already a queen in the same column or diagonal.
2. **solve_nqueens(board, row, n, result)**: This recursive function places queens row by row. It backtracks if placing a queen doesn't lead to a solution.
3. **nqueens(n)**: This function initializes the chessboard and prints all possible solutions.

Output

The output of the solution for n = 4 would look like this:

```
. Q . .
. . . Q
Q . . .
. . Q .


. . Q .
Q . . .
. . . Q
. Q . .
```

In this output:

- Q represents a queen.
- . represents an empty space.
- Each solution is printed in a format where each row corresponds to a row on the chessboard.

Time Complexity

The time complexity of the N-Queens problem using backtracking is exponential. In the worst case, the algorithm needs to explore all N^N possible board configurations, but by pruning invalid solutions early (through the is_safe checks), the number of solutions is significantly reduced.

- **Worst-case time complexity**: O(N!) because for each row, we try placing queens in different columns, but the pruning reduces the number of recursive calls.

Conclusion

The N-Queens problem is an excellent example of how backtracking can be used to find solutions to complex combinatorial problems. This method explores all possibilities, but avoids wasting time on infeasible solutions by using safety checks and backtracking when needed. The solution is efficient for moderate values of N and provides a good exercise in recursion and problem-solving.