N-Queens Problem: A Backtracking Approach

Sumanth

June 18, 2025

Abstract

The N-Queens Problem is a classic constraint satisfaction problem that involves placing N queens on an $N \times N$ chessboard such that no two queens threaten each other. Using a **backtracking** approach, the algorithm systematically explores row-wise queen placements, undoing invalid configurations. This document details the problem formulation, pseudocode, complexity analysis, practical applications, and implementation considerations, including constraint checking and optimizations for large N. The implementation focuses on finding the first valid solution, with notes on extending to all solutions.

1 Introduction

The N-Queens Problem seeks to place N queens on an $N \times N$ chessboard such that no two queens can attack each other. A queen threatens another if they share the same row, column, or diagonal. The **backtracking** algorithm solves this by placing queens row-by-row, checking constraints, and undoing placements that lead to conflicts. This problem is a cornerstone in combinatorial optimization and constraint satisfaction, with applications in puzzles and algorithm design.

2 Problem Formulation

Given an $N \times N$ chessboard, place N queens such that:

- Each row contains exactly one queen.
- No two queens share the same column or diagonal.

The output is a configuration (or all valid configurations) of queen positions, typically represented as an array $\text{col}[i]$, where $\text{col}[i]$ is the column position of the queen in row i.

2.1 Key Idea

The algorithm uses **backtracking** to explore all possible queen placements row-by-row. For each row, it tries placing a queen in each column, checks if the placement is valid (no conflicts with previous queens), and recursively proceeds to the next row. If a conflict arises, it backtracks to try a different column in the current row.

3 N-Queens Algorithm

The algorithm uses a recursive backtracking approach with constraint checking for columns and diagonals. Below is the pseudocode for finding the first valid solution:

Algorithm 1 N-Queens Algorithm (First Solution)

```
Input: Board size N
Output: Array col representing queen positions, or "No solution"
Initialize col[1...N] ← 0                          ▷ Tracks column for each row
function SolveNQueens(row, N)
    if row > N then
        Output col as a solution
        Return true                                ▷ Stop after first solution
    end if
    for j = 1 to N do
        if IsSafe(row, j, col) then
            col[row] ← j
            if SolveNQueens(row + 1, N) then
                Return true
            end if
            col[row] ← 0                           ▷ Backtrack
        end if
    end for
    Return false
end function
Procedure IsSafe(row, col, col):
for i = 1 to row − 1 do
    if col[i] = col or |col[i] − col| = |i − row| then
        Return false
    end if
end for
Return true
Call SolveNQueens(1, N)
```

3.1 Constraint Checking

The `IsSafe` function checks if a queen can be placed at position (row, col):

- **Column Conflict**: No previous queen in the same column ($\text{col}[i] \neq col$).
- **Diagonal Conflict**: No previous queen on the same diagonal ($|\text{col}[i] - col| \neq |i - row|$).

3.2 Finding All Solutions

To find all solutions (as noted by the contributor), modify the base case to continue exploring instead of returning after the first solution:

$$\text{If } row > N, \text{ output col and return false to keep exploring.}$$

4 Complexity Analysis

- **Time Complexity**: $O(N!)$ in the worst case, as the algorithm explores all permutations of queen placements, though pruning reduces the number of invalid branches.
- **Space Complexity**:
  - $O(N^2)$ for the board representation (if explicitly stored).
  - $O(N)$ for the recursion stack and the col array.

5 Practical Applications

The N-Queens Problem is used in:

- **Board Puzzles**: Solving puzzles like Sudoku or chess-based problems.
- **Constraint Satisfaction**: Modeling problems with complex constraints.
- **Combinatorial Optimization**: Studying permutations and configurations.
- **Algorithm Testing**: Benchmarking backtracking and optimization techniques.

6 Example

For $N = 4$, one possible solution to the 4-Queens problem is:

	1	2	3	4
Row 1	.	Q	.	.
Row 2	.	.	.	Q
Row 3	Q	.	.	.
Row 4	.	.	Q	.

This corresponds to col $= [2,4,1,3]$, meaning queens are placed at $(1,2),(2,4),(3,1),(4,3)$. No two queens share a column or diagonal.

7 Limitations and Extensions

- **Limitations**:
  - Slow for large N due to $O(N!)$ complexity.
  - Memory-intensive if storing the full board explicitly.
- **Extensions**:
  - **Bitset Optimization**: Use bit manipulation to track columns and diagonals, reducing space to $O(N)$.
  - **Symmetry Reduction**: Exploit board symmetries to prune equivalent solutions.
  - **Constraint Propagation**: Precompute valid positions to reduce backtracking.

8 Implementation Considerations

- **Board Representation**: Use a 1D array col[N] to store column positions, avoiding the need for an $N \times N$ board.
- **Constraint Checking**: Optimize `IsSafe` by using boolean arrays for columns and diagonals to reduce checking time.
- **All Solutions**: Modify the algorithm to collect all solutions by not terminating after the first valid configuration.
- **Large N**: Use bitsets or heuristic pruning for better performance on large boards.
- **Output Format**: Display solutions as a visual board or coordinate list for clarity.

9 References

- GeeksForGeeks: N-Queens Problem
- Wikipedia: Eight Queens Puzzle
- Cormen, T. H., et al., *Introduction to Algorithms*, Chapter on Backtracking.