Java Project

On

N-Queen Visualizer

Submitted by

V.VIJAY

Registration Number

12205419

SCHOOL OF COMPUTER SCIENCE

Lovely Professional University

Phagwara, Punjab.

Introduction

The N-Queens problem is a well-known combinatorial challenge in computer science and mathematics. The objective is to place N queens on an N×N chessboard such that no two queens threaten each other. This means no two queens can share the same row, column, or diagonal. The problem exemplifies the complexity and elegance of constraint satisfaction problems and has applications beyond chess, such as in parallel processing and optimization problems. This report details the implementation of a Java-based solution using backtracking, accompanied by a visualization tool to illustrate the solving process.

History of the N-Queens Problem

The N-Queens problem was first proposed by the German chess enthusiast Max Bezzel in 1848. Initially posed as an 8-queens problem, it sought a way to place eight queens on an 8×8 chessboard without any two queens attacking each other. The problem quickly attracted the attention of mathematicians and puzzle enthusiasts due to its challenging nature and the insight it offers into combinatorial problems.

Over time, the problem was generalized to the N-Queens problem, where N queens must be placed on an N×N board. Numerous methods have been developed to solve the problem, ranging from simple brute-force algorithms to sophisticated heuristics and optimization techniques. The N-Queens problem remains a popular subject in theoretical computer science and recreational mathematics, highlighting the intersection of algorithmic design, computational complexity, and problem-solving creativity.

Various Solution Methods

Several approaches can be employed to solve the N-Queens problem, each with its advantages and limitations. Here, we discuss four primary methods:

1. Brute Force

The brute-force method involves generating all possible configurations of N queens on the board and checking each configuration to see if it meets the criteria. This approach is straightforward but highly inefficient due to its exponential time complexity, making it impractical for large values of N.

2. Backtracking

Backtracking is a recursive algorithmic technique that incrementally builds candidates for the solutions and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly lead to a valid solution. This method significantly reduces the number of configurations to be checked by pruning invalid configurations early.

3. Constraint Programming

Constraint programming uses constraints to systematically reduce the search space. Constraints are applied to eliminate invalid placements of queens, allowing the solver to focus only on feasible configurations. This approach is more efficient than brute force and can handle larger values of N more effectively.

4. Genetic Algorithms

Genetic algorithms are heuristic search algorithms inspired by the process of natural selection. They use techniques such as selection, mutation, and crossover to evolve a population of solutions over successive generations. While not guaranteed to find a solution, genetic algorithms can often find acceptable solutions quickly for large N.

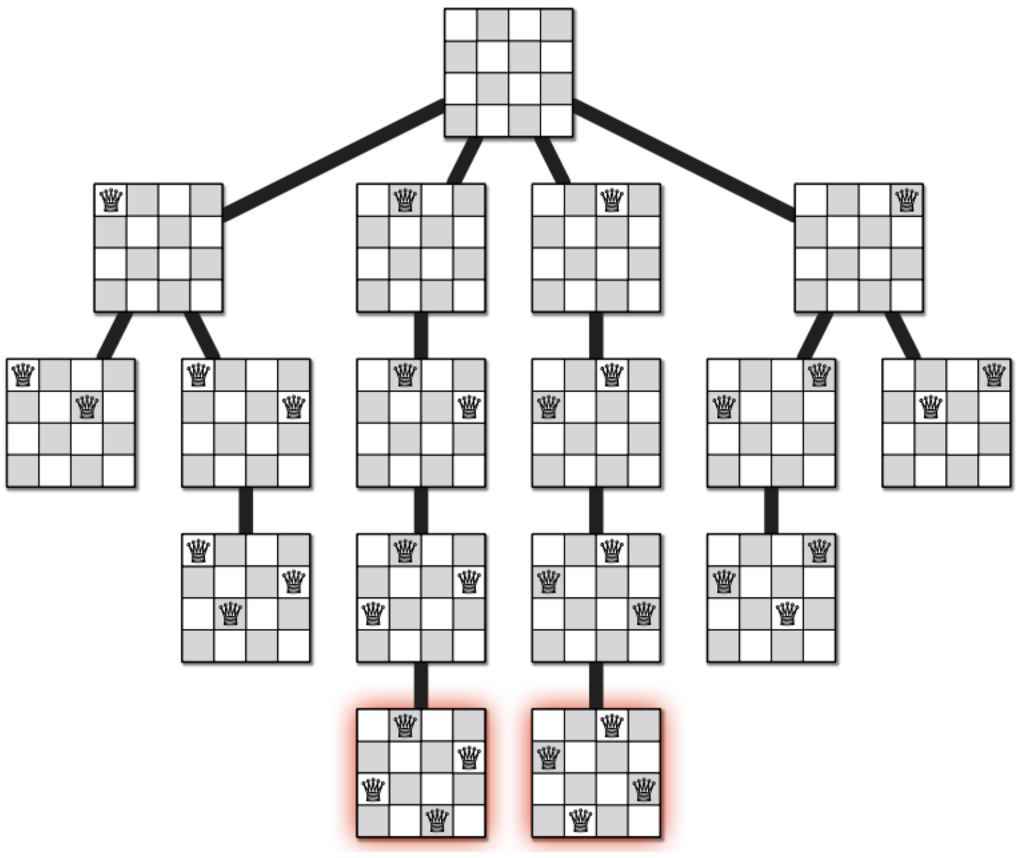
Detailed Backtracking Solution

The backtracking algorithm is a popular and effective method for solving the N-Queens problem. It incrementally builds solutions by placing queens one by one in different rows and backtracks when it encounters a conflict.

Algorithm Description

1. **Initialization**: Start with an empty board and an array to store the positions of the queens.
2. **Recursive Placement**: Place a queen in a row and check if it is safe.
3. **Safety Check**: Ensure that the current placement does not conflict with any previously placed queens.
4. **Backtracking**: If a conflict is found, remove the queen and try the next column. If no valid column is found, backtrack to the previous row.
5. **Solution Storage**: When all queens are placed safely, store the configuration as a solution.
6. **Iteration**: Repeat the process until all possible solutions are found.

Images Description of Solving 4-Queen :

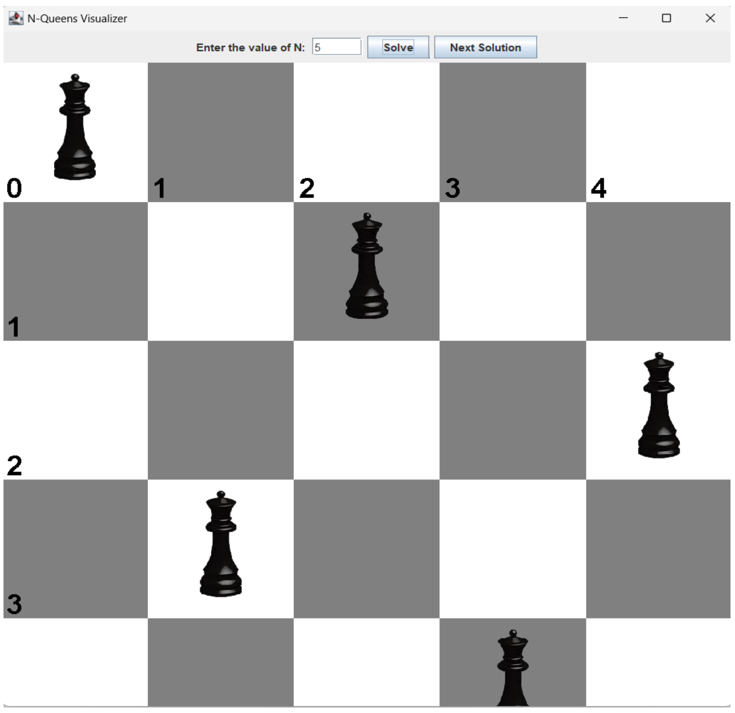


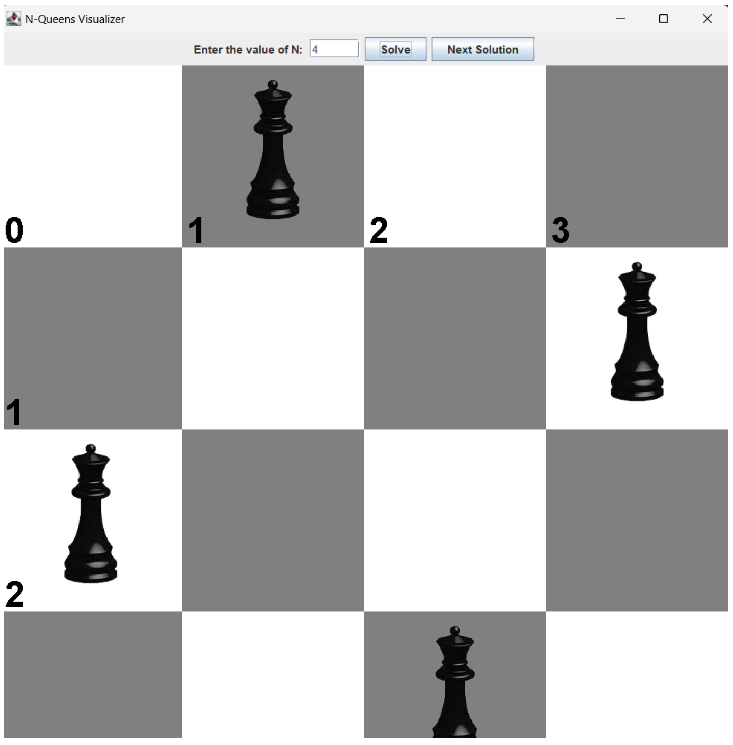
Implementation in Java

The Java implementation involves two main classes: NQueensSolver and NQueensVisualizer. The NQueensSolver class handles the solving process, while the NQueensVisualizer class provides a graphical user interface for visualization.

For the Code : N-Queen-Visualizer_Github

Screenshots of Output:





Time Complexity

The time complexity of the N-Queens problem varies significantly depending on the algorithm used to solve it. Here, we focus on the backtracking algorithm used in our solution.

Brute Force Approach

The brute-force method generates all possible placements of N queens on an N×N board and checks each configuration for conflicts. The number of ways to place N queens on an N×N board is N^N (since each queen can be placed in any of the N columns independently). Checking each configuration for conflicts takes O(N) time. Therefore, the brute-force approach has a time complexity of O(N^N * N) = O(N^(N+1)), which is highly inefficient for large N.

Backtracking Approach

Backtracking is a more efficient method that incrementally builds solutions and abandons configurations that violate the problem's constraints. Let's analyze its time complexity step by step.

1. **Placement of Queens**: At each row, the algorithm attempts to place a queen in each column. This gives us N choices per row.
2. **Safety Check**: For each placement, the algorithm checks whether the queen can be attacked by any previously placed queens. This check involves scanning up to N-1 previously placed queens and verifying they do not conflict. Thus, the safety check takes O(N) time.

Detailed Complexity Analysis

The backtracking algorithm operates as follows:

- For the first queen, there are N possibilities.
- For the second queen, after placing the first queen, there are N-1 possibilities (in the worst case).
- This pattern continues, reducing the possibilities as we go down each row, but since backtracking prunes invalid configurations early, not all these possibilities are explored.

If we do not consider the pruning, the number of configurations to check is NNN^NNN. However, pruning significantly reduces this number. For each level of recursion (each row), the number of valid configurations decreases because invalid configurations are discarded early.

To better understand the backtracking complexity, consider:

- At the first level (first row), there are N choices.
- At the second level, there are approximately N−1N - 1N−1 choices after discarding invalid positions.
- This process continues, with each subsequent level having fewer valid choices.

Thus, the number of operations can be approximated by: T(N)=N×T(N−1)T(N) = N \times T(N-1)T(N)=N×T(N−1) This recursive relation approximates to: T(N)≈N!T(N) \approx N!T(N)≈N!

Upper Bound

The worst-case time complexity of the backtracking algorithm is O(N!). This is significantly better than the O(N^(N+1)) of the brute-force approach. For small values of N, the algorithm runs efficiently. However, as N increases, even O(N!) becomes impractical for very large N due to the factorial growth rate.