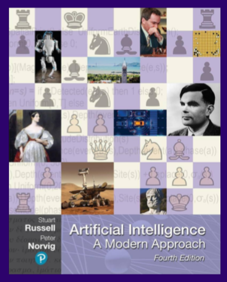

# Infinite N-Queens

Jason Van Humbeck

Wilfrid Laurier University

SCAN TO SEE REPO

github.com/jayfvh/Linear_NQueens

## Problem

A classic Artificial Intelligence (AI) Problem is N-Queens. Here, the goal is to place N Queens on an NxN board in such a position where none are able to attack each other.

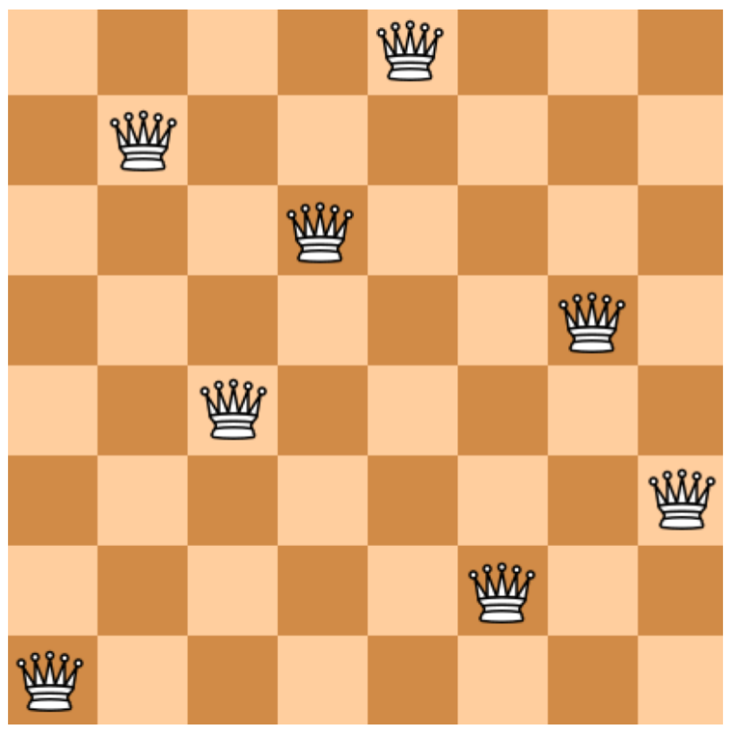

Figure 1. Example: Valid Solution for 8x8 Board

## Problem Size

Finding a solution through pure brute force (trying all possible board configurations) has a runtime complexity of $O(n!)$. This approach is highly inefficient and quickly becomes infeasible for larger boards.

Moreover, storing the board state also consumes significant space, making it both impractical and expensive. At n = 1 Billion, the memory requirement alone would exceed 125 Petabytes!

Basic Optimizations

- A valid board can only have one queen per column, so instead of storing the entire board, we store only the row position of each queen per column. This reduces space complexity from $O(n^2)$ to $O(n)$. Example: The board in Figure 1 can be represented as [7,1,4,2,0,6,3,5].
- Previously, checking conflicts for a given square took O(n) time, but this can be reduced to O(1). By maintaining arrays tracking queen conflicts in each row, left diagonal, and right diagonal, conflicts can be checked in constant time. For any given cell you can determine its left diagonal constraint by (row - column + n - 1) and the right more simply with (row + column).

## Min Conflicts

Instead of blindly guessing board configurations, the **Min Conflicts** algorithm makes informed decisions by selecting placements with the least conflicts. Repeatedly applying this algorithm leads to a solved board, though only a subset of boards can be solved within an efficient time frame.

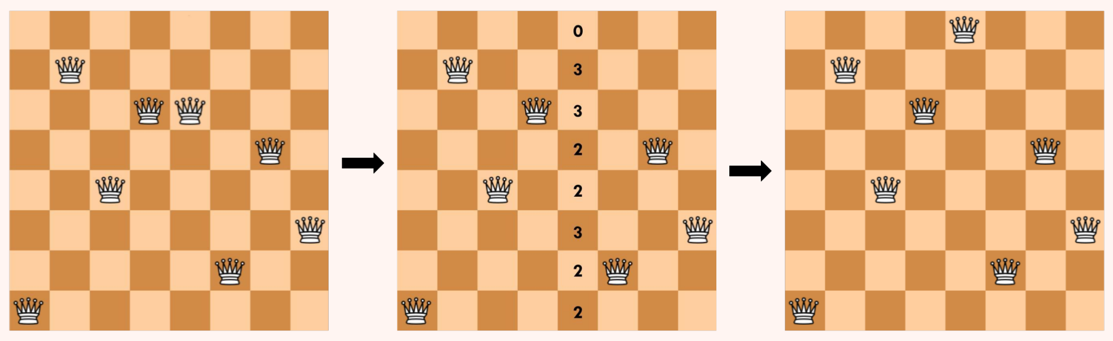

Figure 2. Finding a New Position for a Conflicting Queen Using Min Conflicts

While useful, Min Conflicts becomes impractically slow for larger boards, as it checks all cells in a given column—even those that are clearly unlikely to yield a conflict-free solution. For instance, rows already containing queens need not be checked as they are guaranteed to yield a minimum of 1 conflict.

**Row Queue:** To address this, a circular queue maintains a list of viable rows, rotating through options. This drastically reduces the search space and minimizes redundancy.

However, Even when leveraging Min Conflicts and the Row Queue, alongside other optimizations, some randomly generated boards still lead to extremely long or infinite run-times.

## Golden Boards

For any board size, there exist well-structured configurations—dubbed **Golden Boards**—that can reach a solution within 50 steps. Using these significantly reduces solve times while increasing consistency.

However, randomly generating Golden Boards is unreliable, as their occurrence is rare and non-deterministic. The larger the board, the harder it is to randomly generate one.

## A Greedy Solution

To consistently generate Golden Boards, randomness must be removed. Previously, using Min Conflicts for this task required $O(n^2)$, making it infeasible for large N. However, the Row Queue reduces this to just $O(n)$! This is achieved in three ways:

1. By maintaining a shortlist of rows, the search space is reduced based on knowledge from previously solved columns—cutting it roughly in half on average.
2. The Row Queue enforces a stop condition that limits row attempts per column. This prevents exhaustive searches on difficult columns, deferring them until the full board is populated, allowing more informed decisions. This alone reduces runtime to $O(C \cdot n) = O(n)$
3. Mathematically, certain patterns emerge when building a Greedy Board sequentially (left to right, top to bottom). If column *C* has a queen at row *R*, then column *C* + 1 cannot place queens at *R* + 1, *R* - 1, or *R*. Furthermore, rows between *R* + 2 and *R* + *S* + 2 (where *S* is the stop condition) have a higher probability of yielding valid solutions than those before. Since the Row Queue is circular, it enables easy and automatic rotation of starting search point to *R* + 2, optimizing search space.

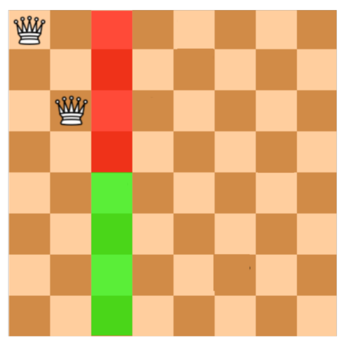

Figure 3. Row Availability in Column 2 After Greedy Placement in 0 1.

As a greedy approach, this method makes the best choice at each step using prior knowledge. While not always perfect, astonishingly, for any N up to infinity, a stop condition of 50 attempts results in a board with only 2-6 conflicts - generally solvable within 50 steps!

## Finding A Solution

Given a greedily-found Golden Board from the previous step, we can now solve it efficiently—usually within 50 steps. Since the Row Queue tracks all used rows, the exhaustive searches at this stage are minimal and easily fall within the stop condition.

The function begins by applying the min-conflicts algorithm to a randomly selected unsolved columns. This slight randomness improves efficiency. If Min Conflicts cannot find a conflict-free (0-conflict) row, the function switches to a modified version of the algorithm that prioritizes 1-conflict solutions, stopping early if one is found, and otherwise considering 2-conflict solutions as a last resort.

After placing a queen, all conflicting columns are reset: they are marked unsolved, and their previous row (if applicable) is re-added to the Row Queue.

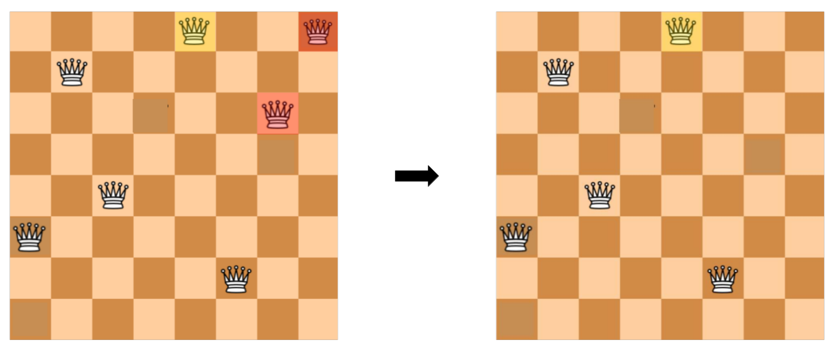

Figure 4. Queen Placement Has 2 Conflicts; The Conflicting Queens Are Therefore Removed

The process then repeats. While this approach almost always converges within 50 steps, a secondary stop condition acts as a safeguard, terminating the solution if it takes too long.

## Checking Validity

Once a solution is found, an external function is used to verify correctness. Given the raw board array, it reconstructs all constraints tallying total conflicts for each. It then iteratively checks all Queens to calculate total conflicts - if any conflicts remain on the board, the function returns "Failure"; otherwise, it confirms "Valid."

## Runtime

With both stopping conditions set < n, the runtime of generating the Greedy Board takes $O(S1 \cdot n)$ and the runtime of Min Conflict Solution is $O(S1 \cdot S2)$ (Where S1 = Min Conflict Stop Condition; S2 = Solution Guard Rail). Therefore the total runtime of N-Queens is $O((n) + (1))$ = $O(n)$. This allows linear runtime results, which could theoretically scale to Infinity:

| N = | Runtime Average(Seconds) |
|---|---|
| 1,000,000 | 0.25s |
| 10,000,000 | 1s |
| 50,000,000 | 4s |
| 100,000,000 | 11s |
| 1,000,000,000 | 160s* |

Table 1. Tests Ran with i7-9700 - 64 GB RAM. Larger N slowed due to physical memory constraints.