N-Queens CSP Solver: Design and Implementation Report

Omar Imamverdiyev, Mehriban Aliyeva

February 24, 2026

Abstract

This report documents the implemented N-Queens project as a practical CSP system, not only as a theoretical model. It explains how the code is organized, how min-conflicts is combined with MRV, LCV, tie-breaking, and AC-3, and how input boards are generated for testing. The focus is on actual control flow and data structures used in the repository.

1 Project Scope

The project solves N-Queens for board sizes in the assignment range $10 \le n \le 1000$. The solver supports two start modes:

- **Random start**: provide `--n`, generate a permutation board.
- **Input start**: provide `--input-file`, load a board from file.

The required CSP components are implemented:

- iterative search (min-conflicts style),
- MRV/LCV with explicit tie-breaking,
- AC-3 constraint propagation over row domains.

2 CSP Model Used by the Code

Each row is one variable:

$$X_i \in \{0, 1, \ldots, n-1\}, \quad i = 0, \ldots, n-1.$$

The value X_i is the column of the queen in row i.

Binary constraints between each pair of rows $i \neq j$ are:

$$X_i \neq X_j, \tag{1}$$

$$|X_i - X_j| \neq |i - j|. \tag{2}$$

This is a complete constraint graph over rows. The implementation stores complete assignments and iteratively repairs conflicts.

3 Code Structure

File	Responsibility
`main.py`	CLI parsing, mode selection, assignment-range check
`nqueens/csp.py`	Compatibility wrapper (`NQueensCSP`)
`nqueens/csp_state.py`	Board state + O(1) conflict counters
`nqueens/min_conflicts.py`	Iterative solver loop and heuristics
`nqueens/ac3.py`	AC-3, `revise`, arc support checks
`nqueens/io_utils.py`	Input parsing with comments/blank-line support
`nqueens/utils.py`	Final board validator (`is_valid`)
`generate_nqueens.py`	Test/input-board generator script

4 Solver Workflow

4.1 Entry Point and Wrapper

`main.py` reads arguments, enforces $10 \leq n \leq 1000$, and builds `NQueensCSP`. The wrapper in `nqueens/csp.py` resets state according to start mode, then calls `solve_min_conflicts`.

4.2 State Representation

`NQueensState` stores:

- `board[row] = col`,
- `col_count[col]`,
- `diag1_count[row - col + n]`,
- `diag2_count[row + col]`.

Conflict evaluation is O(1):

$$\text{conflicts}(r, c) = \text{col\_count}[c] + \text{diag1\_count}[r - c + n] + \text{diag2\_count}[r + c] - 3 \cdot \mathbf{1}[\text{board}[r] = c].$$

The subtraction removes the queen's own current contribution when evaluating its present column.

4.3 Min-Conflicts Loop

At each step:

1. collect conflicted rows,
2. stop if none remain,
3. track stagnation using "best conflicted count" and "stagnant steps",
4. apply restart/noisy escape if stalled,
5. sample conflicted rows and build temporary domains,
6. select row using MRV + tie-break,
7. select value using LCV + tie-break/fallback,
8. move one queen and update counters incrementally.

The algorithm is stochastic in row sampling and exact-tie selection, so runtime can vary across runs.

5 Heuristics and Propagation in the Implementation

5.1 MRV with Conflict-Aware Tie-Break

MRV is applied on sampled conflicted rows, not all rows. For each sampled row:

- domain size is measured after optional AC-3,
- tie-break prefers larger current conflict count,
- exact ties are broken randomly.

This keeps variable choice focused but avoids full-board domain recomputation each iteration.

5.2 LCV on Active Neighborhood

For a chosen row, LCV ranks candidate columns by how many values they eliminate from neighboring sampled rows. The implementation uses fast membership checks for at most three forbidden values per neighbor (same column and two diagonals), then breaks ties by smaller column index.

5.3 AC-3 Integration

Domains are seeded from each row's minimum-conflict columns, capped to a small size, then optionally propagated with AC-3. Propagation is periodic and also triggered during stagnation, instead of running every step.

In `ac3.py`, support testing is optimized: if neighbor domain size is greater than three, support always exists for N-Queens pairwise constraints, so a full nested scan is unnecessary.

6 Adaptive Parameters by Board Size

The solver uses size-based presets to keep per-step work bounded:

Range	`sample_size`	`domain_cap`	`ac3_period`	`stagnation_limit`
$n \geq 400$	$\min(8, n)$	$\min(6, n)$	6	$\max(80, \lfloor n/2 \rfloor)$
$100 \leq n < 400$	$\min(12, n)$	$\min(8, n)$	4	$\max(100, n)$
$n < 100$	$\min(30, n)$	$\min(12, n)$	1	$\max(120, 6n)$

Restart budget is also bounded:

$$\texttt{max\_restarts} = \max\left(4, \min\left(60, \frac{\texttt{max\_steps}}{\max(1, \texttt{stagnation\_limit})}\right)\right).$$

7 Input Handling and Validation

`read_input` accepts one integer per line and ignores:

- blank lines,
- inline comments after `#`.

After parsing, it validates:

- file is non-empty,
- each column is in range $[0, n-1]$,
- columns form a permutation (one queen per column).

8 Generator Script (`generate_nqueens.py`)

8.1 Naming Note

The repository contains `generate_nqueens.py`; there is no file named `generate_npuzzle.py`.

8.2 Purpose

The generator creates input boards for solver experiments and debugging. It writes one column value per line, matching the parser format.

8.3 Mode Logic and Structure

- `--random`: `generate_random_board(n)` uses `random.sample(range(n), n)` to produce a permutation.
- `--easy`: `generate_easy_board(n, attempts=200)` samples many random permutations, scores each with `_conflict_count`, and keeps the lowest-conflict board found.
- `--solution`: `generate_constructive_solution(n)` returns a deterministic even-columns-then-odd-columns ordering (for even n only).
- `--hard-diagonal`: returns $[0, 1, \ldots, n-1]$ (main diagonal).
- `--hard-anti`: returns reversed range (anti-diagonal).

8.4 Conflict Scoring in `_conflict_count`

Instead of pairwise $O(n^2)$ scanning, the function builds column and diagonal frequency arrays, then sums pair counts with:

$$\binom{k}{2} = \frac{k(k-1)}{2}.$$

This is efficient and consistent with the solver's conflict model.

8.5 Practical Caveat

The `--solution` mode name suggests a guaranteed solved board, but the current sequence is only a deterministic permutation pattern and should be treated as a structured start state unless separately validated.

9 Testing Snapshot

Unit tests in `tests/test_nqueens.py` cover:

- file parsing and validation,
- AC-3 primitives (`queens_compatible`, `revise`, `ac3`),
- state counter updates after moves,
- CSP wrapper behavior for random/input start modes.

Current suite status is 10/10 passing.

10 Complexity and Practical Behavior

- Conflict query and move update are O(1).
- Per-step search work is dominated by scanning candidate columns for sampled rows.
- AC-3 adds overhead but improves local pruning when triggered selectively.
- The approach is incomplete (local search), but practical for large n with restart/noise strategies.

11 Conclusion

This implementation follows the CSP framing while prioritizing runtime behavior needed for larger boards. The final design combines compact state counters, adaptive min-conflicts, MRV/LCV with tie-breaking, and selective AC-3. The generator script complements the solver by creating easy, random, and adversarial starts that are useful for evaluation.