SAPIENZA
UNIVERSITÀ DI ROMA

Artificial Intelligence Homework: n-Queens

Comparative Study: Classical Search and CSP

Academic Year 2025/2026

Master in Artificial Intelligence and Robotics

Gaia Scarponi

(1905516)

Professors
Marco Favorito
Francesco Fuggitti

January 2, 2026

Contents

1 Introduction

The n-Queens problem is a fundamental benchmark in Artificial Intelligence for testing search strategies and constraint satisfaction techniques. The objective is to place n non-attacking queens on an $n \times n$ chessboard.

In this work, an entire pipeline of experiments has been implemented, as requested by the assignment:

1. **Modeling**: Defining a memory-efficient state representation.
2. **A* Implementation**: Writing a modular search algorithm from scratch following the guidelines of Slide 32.
3. **CSP Reduction**: Modeling the problem using a declarative paradigm.
4. **Experimental Analysis**: Comparing the scalability of both approaches through empirical metrics, including a study on heuristic overhead.

2 Task 1: Problem Modeling

2.1 State Representation

Instead of a $n \times n$ matrix, a **tuple of integers** is employed. For example, $(0, 2, 1)$ means:

- Queen 1 is in Column 0, Row 0.
- Queen 2 is in Column 1, Row 2.
- Queen 3 is in Column 2, Row 1.

This representation is $O(n)$ in space and ensures that no two queens share the same column by construction. Using Python tuples makes states *hashable*, which is mandatory for the `explored` set in A*.

2.2 State Space Analysis

The theoretical state space is n^n. However, by checking for row and diagonal conflicts during node generation (look-ahead), the *effective* branching factor is drastically reduced. This has been analyzed in the Experimental Results section.

3 Task 2.1: Implementation of A*

3.1 Algorithm Logic

The A* algorithm was implemented with **duplicate elimination** and **no reopening**. Since the cost of placing each queen is constant ($g = 1$) and the heuristic is consistent, the first time a state is reached, the optimal path to it is guaranteed to have been found.

3.2 Heuristic Functions

To guide the A* search, two different heuristic functions were implemented and compared:

3.2.1 Simple Heuristic (h_{simple})

$h_1(s) = n - len(s)$. This heuristic is **perfectly informed** regarding the cost (h^*), as each action has a cost of 1 and exactly $n - len(s)$ more queens must be placed. In practice, this transforms A* into a form of depth-first search because all nodes at the same level have the same f-cost ($f = g + h = n$).

3.2.2 Advanced Heuristic ($h_{advanced}$): Look-ahead Pruning

The second heuristic incorporates a **Look-ahead** strategy inspired by the *Forward Checking* principle. It scans future columns to ensure at least one safe square exists for each.

- **Admissibility**: It remains admissible as it never exceeds the real cost.
- **Pruning**: If a future column has 0 legal moves, the function returns ∞, allowing A* to prune the branch immediately.

3.3 Tie-breaking Strategy

Since many states share the same f-value, the tie-breaking rule in the Priority Queue is critical.

The following logic was implemented within the `Node` class:

```
def __lt__(self, other):
    return self.f < other.f if self.f != other.f else self.h < other.h
```

By choosing the node with the **smaller** h during a tie, the algorithm has been forced to prefer deeper nodes (closer to the goal), which is optimal for the n-Queens problem structure.

4 Task 2.2: Reduction to CSP

The second technique used is the reduction to a Constraint Satisfaction Problem. This approach is declarative: the solution properties are defined, rather than the steps to achieve it.

4.1 Variables and Constraints

n variables $V_0 \dots V_{n-1}$ have been defined, each representing a column. The domain for each variable is $\{0 \dots n-1\}$ (the possible rows). The constraints are:

1. **AllDifferentConstraint**: No two queens in the same row.
2. **DiagonalConstraint**: $|Row_i - Row_j| \neq |i - j|$.

The CSP solver leverages **Arc Consistency** (AC-3), which is proactively superior to the reactive nature of state-space search.

5 Task 3: Experimental Results

Experiments were conducted for n from 4 to 14. All artifacts are stored in the `Outputs/` directory.

5.1 Performance Metrics

Table 1 summaries the execution metrics.

N	A*(Sim)[s]	A*(Adv)[s]	CSP[s]	Nodes (Sim)
4	0.00005	0.00005	0.00011	8
6	0.00022	0.00018	0.00034	53
8	0.00026	0.00054	0.00114	60
10	0.00057	0.00116	0.00069	111
12	0.00222	0.00247	0.00199	332
13	0.00080	0.00152	0.00370	111
14	0.01349	0.01773	0.00285	1870

Table 1: Comparative performance metrics.

5.2 Execution Time and Complexity Analysis

- **The Overhead Trade-off**: While the Advanced Heuristic reduces the number of nodes expanded (see Fig. 2), it is often slower in execution time. This illustrates the *Heuristic Trade-off*: the computational cost of look-ahead pruning at every node exceeds the time saved by exploring fewer nodes for $N \leq 14$.
- **The "Luck" Factor (N=13)**: A notable anomaly occurs at $N = 13$, where the search is faster than at $N = 12$. This is due to the solution density; the DFS-ordered tie-breaking finds a valid leaf almost immediately, proving that exploration order is as vital as heuristic quality.
- **CSP Scaling**: The CSP solver (red line in Fig. 1) shows the most stable growth. For $N = 14$, it outperforms A* by an order of magnitude, benefiting from proactive domain pruning.

5.3 Visual Analysis and Empirical Observations

In this section, a qualitative and quantitative evaluation of the experimental results was provided through the analysis of the generated plots and the final solution for the highest tested N.

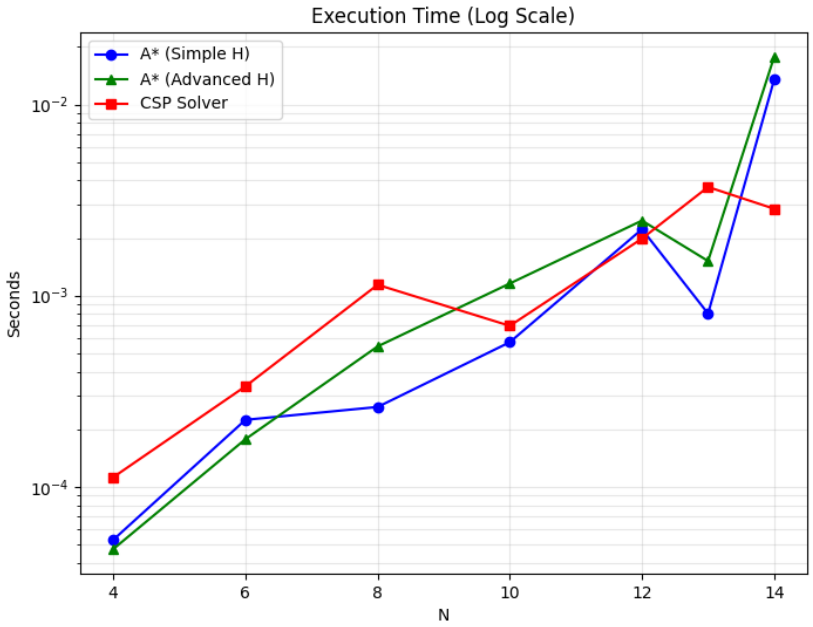

Figure 1: Execution Time comparison (Logarithmic Scale).

5.3.1 Execution Time and Heuristic Overhead

Figure 1 illustrates the execution time on a logarithmic scale. Several key trends emerge:

- **The Heuristic Trade-off**: A counter-intuitive result is visible: A* with the *Advanced Heuristic* (green line) often takes more time than the *Simple* version (blue line), despite being more informed. This is a classic example of **computational overhead**: the time spent at each node to calculate the look-ahead pruning is greater than the time saved by expanding fewer nodes for $N \leq 14$.
- **CSP Efficiency**: The CSP Solver (red line) shows the most stable performance as N increases. For $N = 14$, it outperforms A* by nearly an order of magnitude, demonstrating that *Arc Consistency* is more effective than state-space search for high-dimensional constraint problems.

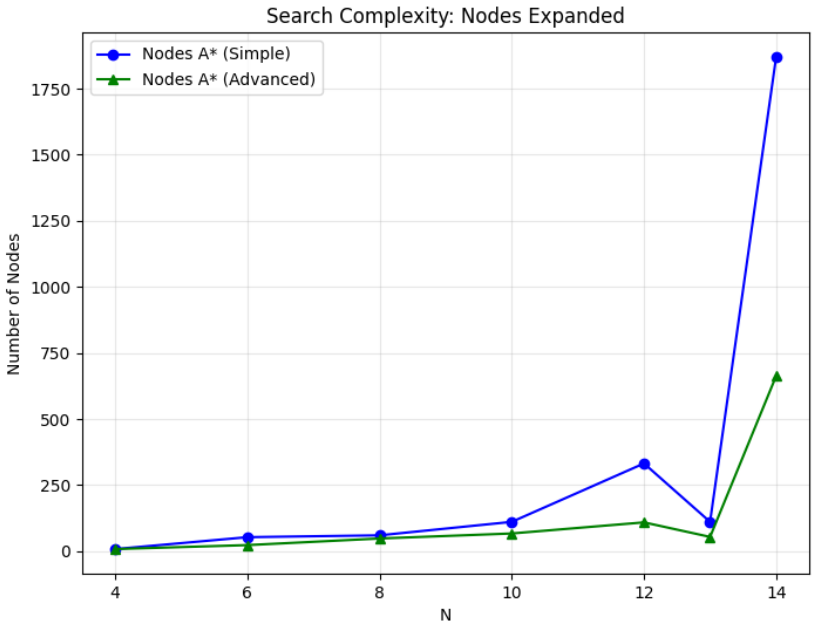

Figure 2: Search Complexity: Comparison of nodes expanded by A* variants.

5.3.2 Search Complexity and the $N = 13$ Anomaly

Figure 2 focuses on the search effort.

- **Pruning Power**: The Advanced Heuristic (green line) successfully expands significantly fewer nodes compared to the Simple Heuristic (e.g., 650 vs 1850 at $N = 14$). This confirms that the look-ahead strategy effectively prunes invalid branches early.
- **The Solution Density Factor**: A notable "dip" is observed at $N = 13$. This is not a measurement error but a property of the n-Queens problem: for certain values of N, valid solutions are more "dense" or easier to find with the specific tie-breaking and expansion order used (DFS-like), allowing the search to reach a goal state with much less exploration.

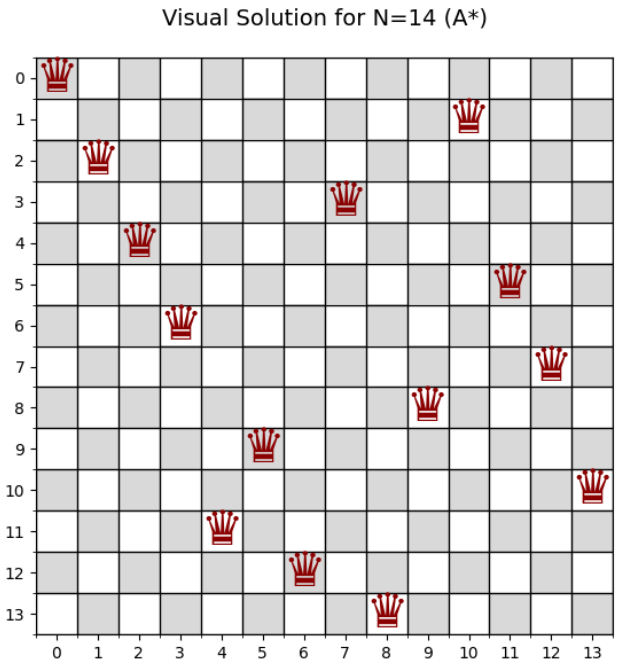

Figure 3: Visual solution for $N = 14$ generated by the A* algorithm.

5.3.3 Solution Feasibility

Figure 3 presents the visual output for the largest experiment ($N = 14$). The board configuration confirms the correctness of the implementation: no two queens share the same row, column, or diagonal, proving that the search correctly identifies valid goal states in a high-dimensional state space.

6 How to Run

6.1 Dependencies

The project requires Python 3.8+ and the following libraries:

- `python-constraint`: for the CSP task.
- `matplotlib`: for generating the charts.

6.2 Execution

To run the full suite of experiments and generate the report data:

```
pip install python-constraint matplotlib
python3 main.py
```

The outputs will be generated in the `Outputs/` folder.

7 Discussion: Search vs. CSP

Beyond numerical performance, this project allowed for a qualitative comparison of the two paradigms:

- **Pruning Power**: A* is "reactive" (it sees a conflict only when it tries to place a queen). CSP is "proactive" (it deduces conflicts before they happen via AC-3).
- **Implementation Effort**: A* required building the entire search infrastructure (priority queue, node management), while CSP shifted the effort to the modeling phase.

8 Conclusion

The project demonstrates that while A* is intuitive, its performance is heavily tied to heuristic quality and node overhead. For combinatorial problems like n-Queens, CSP solvers leveraging Arc Consistency are significantly more efficient as the problem scales.