AI Homework - A* and CSP for N-Queens problem

Vittorio Pisapia - 1918590

January 11, 2026

1 Introduction

This report was produced as part of an Artificial Intelligence homework assignment. The chosen problem is the n-Queens problem, which is addressed using two different AI techniques: A* search and a reduction to a Constraint Satisfaction Problem (CSP).

As part of this work, a complete and modular implementation of the A* algorithm was developed, capable of solving different search problems through suitable problem definitions and heuristics. In addition, the n-Queens problem lends itself naturally to a CSP formulation. For this reason, a CSP-based solution was implemented using the OR-Tools solver. Due to the nature of the problem, the CSP-based approach is expected to scale significantly better than the search-based approach and to be able to solve much larger instances efficiently.

2 Task 1: Problem

The n-Queens problem consists of placing n queens on an $n \times n$ chessboard such that no two queens attack each other. Two queens are said to attack each other if they share the same row, column, or diagonal.

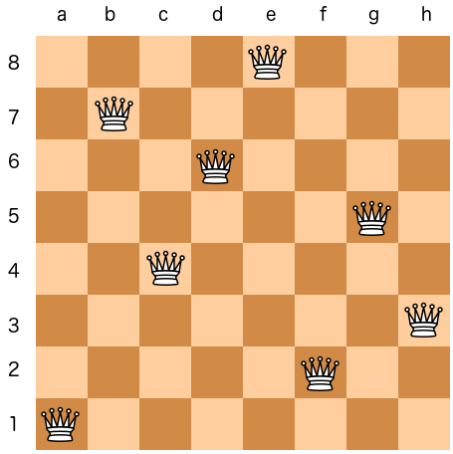

Figure 1: Solution of the standard problem with n=8

3 Task 2.1: Implementation of A*

A* is a classical state-space search algorithm. Its main characteristic is the ordering of the frontier according to the evaluation function

$$f(s) = g(s) + h(s),$$

where $g(s)$ denotes the cost to reach state s from the initial state, and $h(s)$ is a heuristic estimate of the remaining cost to reach a goal state.

3.1 Data structures

Unexplored states (the frontier) are stored in a heap-based priority queue ordered by increasing $f(s)$. A dictionary is used to keep track of the best known cost $g(s)$ for each visited state, while explored states are stored in a set in order to avoid duplicate expansions.

3.2 Algorithm

Initially, the initial state s_0 is inserted into the frontier with $g(s_0) = 0$ and priority $f(s_0) = h(s_0)$. The algorithm then iteratively performs the following steps while the frontier is not empty:

- Extract the state with the lowest $f(s)$ from the frontier; if it is a goal state, the search terminates.
- Otherwise, add the state to the explored set.
- Generate all successor states.
- For each successor, skip it if it already belongs to the explored set.
- Compute the tentative cost to reach the successor as the cost of the current state plus the action cost.
- If the successor has not been visited before, or if a cheaper path is found, update its cost in the dictionary.
- Compute the successor's evaluation value $f(s)$ and insert it into the frontier.

During the search, several performance metrics are collected, including execution time, number of expanded and generated states, minimum, maximum, and average branching factor, and the maximum number of nodes stored in memory at any point during the execution.

The implementation is fully modular and problem-independent. In particular, the problem must provide the following components: a goal test function `is_goal`(s), a successor generation function `successors`(s), a heuristic function `heuristic`(s), and a state embedding function `state_key`(s) used for duplicate detection.

3.3 Problem embedding

For the n-Queens problem, a state is represented as a tuple of column indices corresponding to the rows already filled. For example, the state $(1, 3, 5)$ indicates that queens have been placed in rows 0, 1, and 2 at columns 1, 3, and 5 respectively. The number of placed queens is given by `len(state)`.

A state is considered a goal state when `len(state)` $= n$, meaning that queens have been placed in all rows. Successor states are generated by placing a queen in the next row, trying all possible columns and retaining only those placements that do not introduce conflicts. The cost for every actions is 1.

3.4 Heuristics

Two heuristics were implemented. The first heuristic is defined as the number of remaining rows to be filled,

$$h_1(s) = n - \texttt{len}(s),$$

which is admissible but weak, as it does not distinguish between different states at the same search depth.

The second heuristic takes into account the number of legal column positions available for placing a queen in the next row. States for which the next row admits only a few legal moves are considered more constrained and are assigned a higher heuristic value, while states with many legal options are preferred.

By penalizing states that are closer to dead ends, this heuristic effectively guides the search toward more promising regions of the state space and significantly reduces the number of expanded nodes. However, since the heuristic may overestimate the true remaining cost to reach a goal state, it is not admissible. In the context of the n-Queens problem, this loss of admissibility is reasonable, as all valid solutions have the same cost n.

Formally, let $k(s)$ denote the number of legal column positions available for placing a queen in the next row given state s. States for which $k(s) = 0$ are assigned a very large heuristic value (dead penalty M), effectively postponing their expansion. For all other states, the heuristic is defined as a function of both the remaining number of rows and the number of available moves, assigning lower values to less constrained states.

Let $r = \texttt{len}(s)$ be the number of rows already filled. The second heuristic is defined as:

$$h_2(s) = \begin{cases} 0, & \text{if } r = n, \\ M, & \text{if } k(s) = 0, \\ (n - r) \cdot C + \left\lfloor \dfrac{C}{k(s)} \right\rfloor, & \text{otherwise,} \end{cases}$$

where C is a positive scaling constant.

4 Task 2.2: Implementation of CSP Reduction

In the CSP approach, the n-Queens problem is modeled by introducing one integer variable Q_r for each row r of the chessboard. The value of each variable represents the column position of the queen placed in that row. Each variable has domain $D = \{0, 1, 2, \ldots, n - 1\}$.

4.1 Constraints

Column conflicts are avoided by imposing an all-different constraint over the variables Q_r, ensuring that no two queens are placed in the same column.

Two queens attack each other diagonally if either of the following conditions holds:

$$\begin{aligned} Q_{r_1} - r_1 &= Q_{r_2} - r_2 \quad \text{(main diagonal)} \\ Q_{r_1} + r_1 &= Q_{r_2} + r_2 \quad \text{(anti-diagonal)} \end{aligned}$$

To enforce these constraints, auxiliary variables are introduced to represent the main and anti-diagonal indices for each queen. An all-different constraint is then imposed on both sets of diagonal variables, ensuring that no two queens share the same diagonal.

4.2 Solution

Once the problem and its constraints are defined, a solution is obtained by using the OR-Tools CP-SAT solver, which searches for a feasible assignment of the variables satisfying all constraints. If a feasible solution is found, the values of the variables are extracted to reconstruct the final placement of the queens on the chessboard.

5 Task 3: Experimental Results

This section presents the experimental evaluation of the proposed approaches. All experiments were executed on the same machine using the same implementation.

5.1 A* Experiments

A* was evaluated on instances with $n \in \{4, 6, 8, 10, 12, 13\}$ using both heuristics. Table 1 reports the results of these experiments.

For small problem sizes ($n = 4, 6, 8, 10$), both heuristics achieve very low runtimes, remaining well below one second. However, even in this range, the baseline heuristic expands and generates a significantly larger number of nodes compared to the MRV-based heuristic.

As the problem size increases, the difference between the two heuristics becomes more pronounced. For $n = 13$, the MRV-based heuristic still finds a solution in a few milliseconds, whereas the baseline heuristic requires approximately 47 seconds and expands several million nodes. This highlights the importance of effective heuristic guidance in controlling the growth of the search space.

Additional experiments were conducted using the MRV-based heuristic. The largest instance that could be solved without exhausting system memory was $n = 22$, for which a solution was found in approximately 47 seconds.

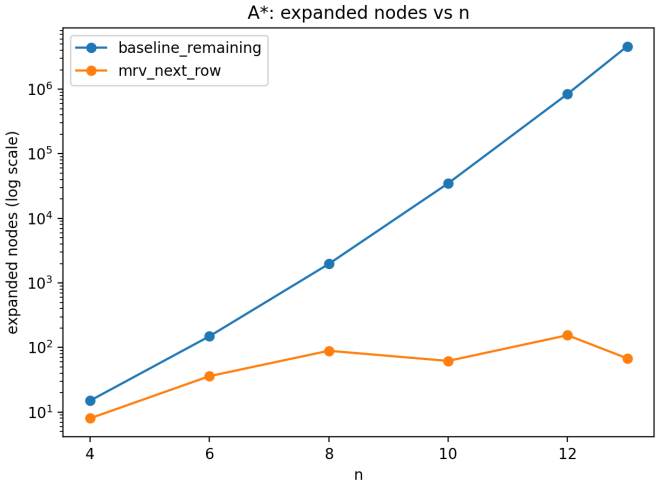

Figure 2: Number of expanded nodes as a function of the board size n

Despite these differences, the average branching factor remains similar for both heuristics.

Even when considering the largest chessboard size that could be solved ($n = 22$), this value remains relatively small for the n-Queens problem, indicating that a search-based approach is not well suited for this application.

Table 1: A* performance comparison between baseline and MRV heuristics

n	Heuristic	Runtime (s)	Expanded	Generated	Avg. Branching
4	Baseline	0.00010	15	16	1.07
4	MRV	0.00007	8	13	1.63
6	Baseline	0.00075	149	152	1.02
6	MRV	0.00026	36	58	1.61
8	Baseline	0.00940	1 965	2 056	1.05
8	MRV	0.00082	89	153	1.72
10	Baseline	0.21382	34 815	35 538	1.02
10	MRV	0.00089	62	118	1.90
12	Baseline	7.07063	841 989	856 188	1.02
12	MRV	0.00261	155	288	1.86
13	Baseline	46.98377	4 601 178	4 674 889	1.02
13	MRV	0.00135	68	150	2.21
22	MRV	47.1636	1168645	1831032	1.57

5.2 CSP Experiments

The CSP-based formulation was evaluated on instances with $n \in \{4, 6, 8, 20, 50, 100, 150, 200\}$. The results are reported in Table 2.

For small and medium board sizes, the CSP approach consistently finds solutions in a fraction of a second. In particular, instances up to $n = 100$ are solved in under one second, demonstrating excellent scalability compared to the search-based approach.

As the board size increases, the solver requires a larger amount of internal search, as reflected by the growth in the number of branches and conflicts. Nevertheless, instances with $n = 150$ and $n = 200$ are still solved successfully within approximately 8 and 12 seconds, respectively.

Overall, these results clearly show that the CSP-based approach scales significantly better than A*, enabling the solution of much larger instances of the n-Queens problem within reasonable time limits.

6 Conclusion

In this work, the n-Queens problem was addressed using two different artificial intelligence techniques: heuristic search with A* and constraint satisfaction using a CSP solver. Although A* can

Table 2: CSP (OR-Tools CP-SAT) performance on the n-Queens problem

n	Found	Runtime (s)	Conflicts	Branches
4	True	0.00842	0	0
6	True	0.00697	4	244
8	True	0.00782	15	478
20	True	0.02608	0	420
50	True	0.15761	0	308
100	True	0.80751	0	820
150	True	8.04443	3 730	249 414
200	True	11.68684	5 472	134 385

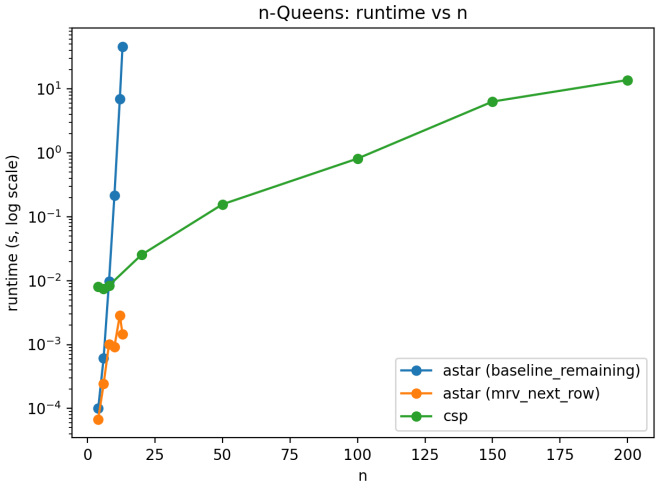

Figure 3: Runtime comparison of A* (baseline and MRV heuristics) and CSP approaches as a function of the board size n

be successfully applied to small instances of the problem, its scalability is strongly dependent on the quality of the heuristic function. Even with an improved heuristic, the search-based approach becomes impractical for larger board sizes due to the exponential growth of the search space.

On the other hand, the CSP formulation proved to be significantly more effective, allowing the solution of much larger instances within reasonable time limits. These results indicate that, for problems such as n-Queens where strong structural constraints exist, constraint-based methods are more suitable than classical state-space search.

7 How to Run

The code requires Python version 3.9 or higher and was tested using Python 3.11.

The external Python libraries required to run the project are:

- OR-Tools
- Matplotlib (required for plotting experimental results)

The project is organized into multiple files:

- `astar.py`: contains the full implementation of the A* search algorithm.
- `nqueens_problem.py`: provides the state-space formulation of the n-Queens problem used by the A* solver.
- `nqueens_csp.py`: defines the CSP formulation of the n-Queens problem and its constraints using OR-Tools.
- `main.py`: allows the user to run a single experiment by selecting the solution method (A* or CSP), the board size n, and optionally printing the solution board to the terminal.

- `benchmark.py`: executes multiple experiments with different board sizes and stores the results in a CSV file named `results.csv`.
- `plot_result.py`: generates plots such as Figures 2 and 3 from the CSV file.