SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF INGEGNERIA INFORMATICA, AUTOMATICA E GESTIONALE

Homework Report: N-queens problem

ARTIFICIAL INTELLIGENCE

Professor:
Marco Favorito
Francesco Fuggitti

Student:
Francesco Morazio
2018794

Academic Year 2025/2026

Contents

1 Introduction

The N-Queens problem is a classic combinatorial challenge in the field of Artificial Intelligence and computer science. The objective is to place N non-attacking queens on an $N \times N$ chessboard such that no two queens share the same row, column, or diagonal. As N increases, the state space grows exponentially, making the problem a significant benchmark for evaluating the efficiency and scalability of different algorithmic paradigms.

In this report are explored and compared two fundamentally different approaches to solve the N-Queens puzzle:

1. **State-Space Search (A* Algorithm):** A pathfinding approach where the problem is treated as a search for an optimal state. We implement a custom heuristic function based on the number of attacking pairs to guide the search from a random initial configuration toward a conflict-free goal.

2. **Constraint Satisfaction Problem (CSP):** A logic-based approach where the problem is modeled as a set of variables with specific domains and global constraints. This method utilizes the OR-Tools CP-SAT solver, which uses advanced constraint propagation and inference techniques.

The experimental results reveal a stark contrast between the two methods. While the A* algorithm is effective for smaller dimensions, it encounters a "computational wall" as N approaches 30 due to the exponential explosion of the branching factor of the generated tree that explores state space. In contrast, the CSP solver demonstrates really good scalability, resolving boards up to $N = 300$ with high efficiency. Our analysis highlights that while both methods are reliable in finding valid solutions, the CSP paradigm is much better for managing the complexity inherent in large-scale versions of this problem.

2 Problem description

The N_queens problem is defined by the requirement to arrange N queens on a grid of $N \times N$ cells such that the final configuration is 'attack-free.' The queen in the chess game is capable of moving any number of squares vertically, horizontally, or diagonally. So, for a placement to be considered admissible (a valid solution), it must satisfy three fundamental constraints simultaneously:

1. **Row Constraint**: no two queens may occupy the same row.
2. **Column Constraint**: no two queens may occupy the same column.
3. **Diagonal Constraint**: no two queens may lie on the same diagonal line, regardless of the direction.

Any configuration that violates even one of these rules is considered inadmissible due to a 'collision' or 'attack.' From a computational perspective, the challenge lies in the fact that as N grows, the number of possible ways to arrange the queens increases factorially, while the number of valid solutions remains relatively small. Therefore, solving the problem requires a systematic way to explore the board and identify positions that respect all constraints without manually checking every possible combination.

A crucial step in solving the N_queens problem is the reduction of the search space through a structured representation. By pre-defining that each queen must occupy a unique column, the problem shifts from searching every possible square on the board to finding a valid permutation of rows. In a naive approach, placing N queens on an $N \times N$ board would result in $\binom{N^2}{N}$ possible combinations, a number that grows a lot even for small values of N. However, by enforcing this constraint and assigning exactly one queen to each column, the search space is reduced to $N!$.

2.1 High level description of the implementation

The constraint about the columns, simplified also the encoding of the board for the problem. Instead of using a matrix of size $N \times N$, an array of size N is used. In this way, each cell of the board can be represented using:

- the index of the array as the column of the cell;
- the value stored in the array as the row.

So the quantity of memory used to store the boards is N times smaller respect to the matrix representation. The representation of each state is saved in the Board class, where the two fields `Board.queens` and `Board.N` store the array that contains the board and the size of the board, that is also the size of the problem. Then the other functions of the class are: `Board.heuristic()` that computes the value of the heuristic for the current state and `Board.get_successors()` that computes all the successors from the current state. Note that this functions will return all the possible successors, but then in the frontier of the A* algorithm, not all the successors are going to be considered.

3 Implementation of A*

The implemented A* algorithm follows a graph-search approach to navigate the N_queens state space, ensuring both completeness and optimality. As required the algorithm was implemented with duplicate elimination and no re-opening. The core of the implementation relies on three primary data structures: a **Priority Queue** (utilizing Python's heapq library) to manage the frontier, a **Dictionary** to track the lowest known cost (g) for states currently in the frontier, and a **Set** to maintain the explored states. The frontier is ordered by the evaluation function $f(n) = g(n) + h(n)$, where:

- $g(n)$ represents the path cost from the initial state and
- $h(n)$ is the heuristic value, so the number of conflicts.

To handle duplicates efficiently, the algorithm employs a 'lazy removal' strategy: if a state is reached via a shorter path, the new cost is updated in the dictionary and a new entry is pushed to the priority queue; redundant entries with higher costs are then discarded during the extraction phase if they no longer match the minimum cost stored in the dictionary. This architecture ensures that the solver correctly identifies the goal state (where $h(n) = 0$) while significantly pruning the search tree.

The cost of each action was considered unitary, since all the moves along the same column are similar (there is no difference in moving a queen by one cell or $N-1$ cells), so the cost of each patch $g(n)$ correspond to the number of queens moved to reach the current state. The `heuristic()` function computes the number of collisions between queens in the current state, so the number of couples of queen that either are on the same row, or are on the same diagonal. The goal for this problem is to reach a state with $h(n) = 0$, so one of the states without collisions.

4 Implementation of CSP

A Constraint Satisfaction Problem (CSP) is a mathematical framework used to define problems as a set of objects whose state must satisfy a number of constraints or limitations. Formally, a CSP is defined by a triple $\langle V, D, C\rangle$:

- V is a set of variables, $\{V_1, ..., V_n\}$.
- D is a set of domains, $\{D_1, ..., D_n\}$, where each D_i contains the allowed values for variable V_i.

- C is a set of constraints, $\{C_1, ..., C_m\}$, which specify allowable combinations of values for subsets of variables.

A solution to a CSP is an assignment of a value to every variable from its respective domain such that all constraints are simultaneously satisfied. This approach is particularly powerful in Artificial Intelligence because it allows for a declarative representation of the problem, where the focus is on what must be achieved rather than how to search for it.

Applying the CSP framework to the N_queens problem allows for a highly optimized search process. Using the simplification previously discussed, the problem can be formally mapped as follows:

- Variables V: a set of N variables $\{Q_1, Q_2, ..., Q_N\}$, where each Q_i represents the queen placed in the i-th column.
- Domains D: for each variable Q_i, the domain D_i consists of integers $\{0, 1, 2, ..., N-1\}$, representing the possible row positions.
- Constraints C: the constraints are defined by the requirement that for any two variables Q_i and Q_j (where $i \neq j$):
  - $Q_i \neq Q_j$ (to ensure they are in different rows).
  - $|Q_i - Q_j| \neq |i - j|$ (to ensure they are not on the same diagonal).

By formalizing the problem this way, we can move beyond simple state-space search and employ CSP-specific techniques such as Constraint Propagation or Backtracking with Inference, which can prune the search tree by identifying impossible assignments before they are even attempted. As suggested the Python OR-tools library is used to solve the problem.

4.1 Different input for the two problems

An important distinction between the search-based approach (A*) and the Constraint Satisfaction Problem (CSP) framework lies in the nature of the input. In the previous A* implementation, the algorithm required a specific initial state (a partial or full board configuration) to begin its exploration of the state space. Instead, the CSP solver does not require an initial arrangement of queens; it only needs the dimension N as input. Since the CSP treats the problem as a set of variables with empty domains that must be filled according to global constraints, the 'starting point' is inherently the empty $N \times N$ board. The solver then systematically assigns values to variables $Q_1, \ldots, Q_N$ based on the constraints, moving the focus from transforming a specific input state to finding any assignment that satisfies the mathematical requirements of the problem.

5 Experimental Results

The experimental results demonstrate that while the A* algorithm is effective for smaller board dimensions, it faces significant challenges as N increases, primarily due to the exponential growth of the state space. A critical bottleneck observed is the branching factor, which exceeded 800 at $N = 30$, forcing the algorithm to evaluate an high number of potential moves for every node expansion. then, the algorithm shows high sensitivity to the initial configuration; for instance, at $N = 18$, the execution time varied drastically depending on how many conflicts were present in the starting state. So, it can be concluded that A* is well-suited for "local repair" tasks where $N < 20$, but it lacks the scalability required for larger problems. For the N-Queens puzzle, $N = 30$ represents a practical "computational wall" for this approach, beyond which the search effort becomes inefficient compared to alternative paradigms.

Test ID	**Dimension N**	**Nodes Visited**	**Path Cost (g)**	**Branching Factor**	**Time (s)**
test1	4	4	3	9.000000	0.000110
test2	4	8	4	7.875000	0.000142
test3	5	5	4	16.200000	0.000174
test4	6	66	4	17.409091	0.003640
test5	8	14	8	47.714286	0.002239
test6	8	64	9	42.640625	0.010482
test7	10	30	10	79.133333	0.009745
test8	12	253	4	96.110672	0.223869
test9	14	77	13	158.402597	0.094566
test10	15	293	15	180.846416	0.526503
test11	16	141	18	211.000000	0.342124
test12	18	7	2	290.142857	0.022494
test13	20	92	17	351.576087	0.499358
test14	22	549	20	412.865209	4.224690
test15	24	537	26	499.346369	5.603158
test16	25	445	27	548.964045	5.552493
test17	26	301	23	594.920266	4.427281
test18	28	2300	25	681.375217	45.425243
test19	30	720	27	807.150000	18.495639
test20	30	955	25	804.843979	24.483518

Table 1: Metrics Summary for A* Algorithm

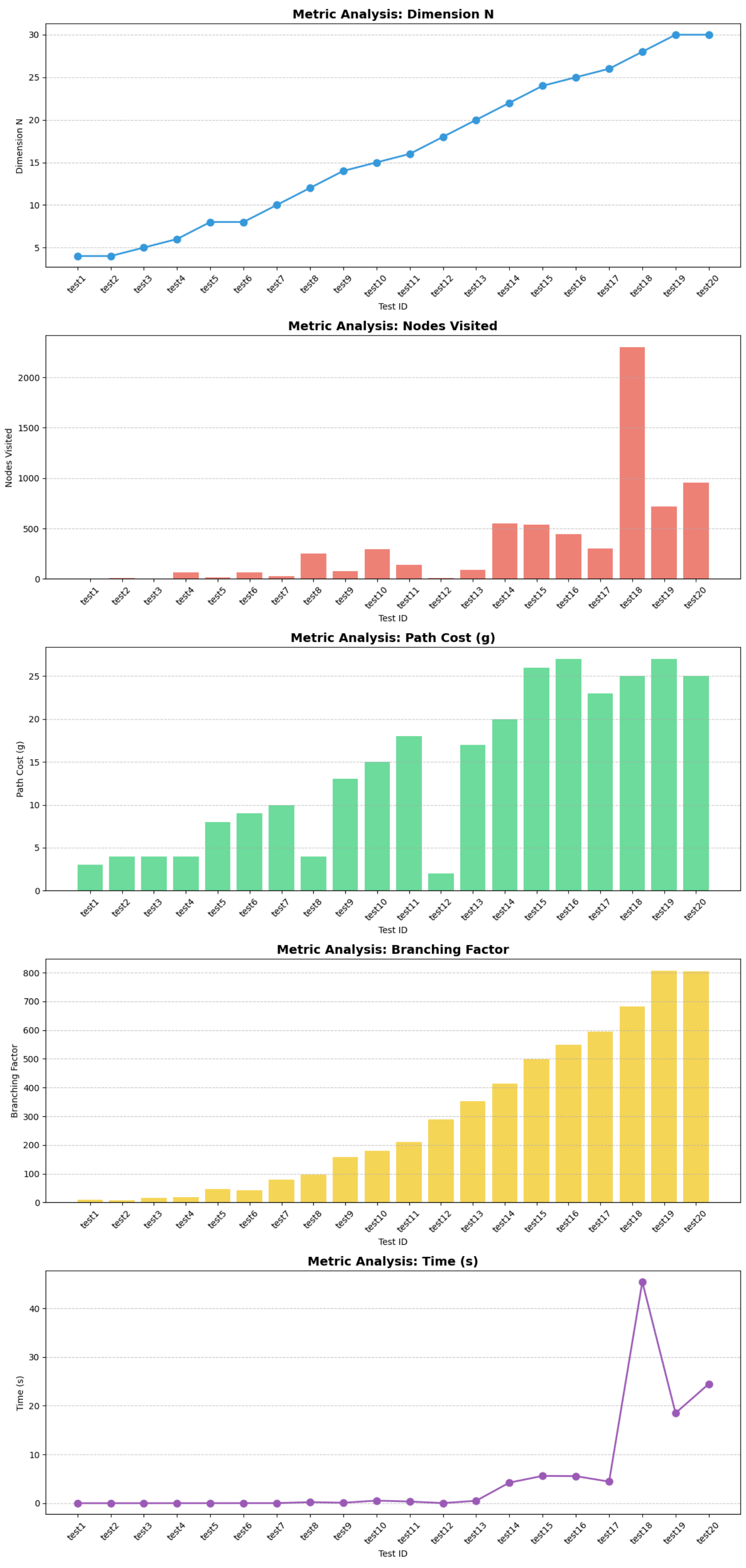

Figure 1: A* metrics graph

The Constraint Satisfaction Problem (CSP) approach, implemented via the OR-Tools CP-SAT solver, demonstrated a radical improvement in both efficiency and scalability. Unlike A*, the CSP solver maintained near-instantaneous execution times across all shared test dimensions, resolving the $N = 30$ case in approximately 0.025 seconds. A key factor in this performance is the power of Constraint Propagation; for most experiments up to $N = 100$, the solver reported zero internal conflicts. This indicates that the global constraints (`AllDifferent`) were able to prune the search space so effectively that a valid solution was found without any backtracking. Even when pushed to an extreme stress test of $N = 300$, the solver managed to navigate over 600,000 branches and find a solution in under 100 seconds. Also the presence of zero internal conflicts in many test cases highlights the effectiveness of the CP-SAT solver's Constraint Propagation and Preprocessing stages. For board sizes up to $N = 30$, the solver is often able to prune the search space so effectively that it reaches a feasible solution in a single 'descending' path, without ever encountering an inconsistent state that would require backtracking. These results conclude that the CSP paradigm is far superior for this class of problem, as it transforms a brute-force search into a process of logical inference, bypassing the exponential complexity that limits state-space search algorithms.

To show how better the CSP performs on this problem, 5 more tests were added only for the CSP. Except for the last test, that had a size 10 times bigger than the highest test for A*, all the others returned a solution in less than 1 second. This is an incredible result in terms of time spendt to find a board configuration without collision. One last experiment has been done, but not reported in the code, trying the A* algorithm with N=50 and all the queens placed in row 0. After 15 minutes, no solution was returned and the notebook couldn't go over that cell, so it was canceled from the notebook. Anyway it is such an important result, because it took more time than the stress test on CSP and didn't return a solution.

Test ID	**N Dimension**	**Time (s)**	**Conflicts**	**Branches**
test1	4	0.002679	0	0
test2	4	0.002697	0	0
test3	5	0.003038	0	125
test4	6	0.002718	0	0
test5	8	0.004286	0	285
test6	8	0.004347	9	475
test7	10	0.005579	8	756
test8	12	0.006569	0	533
test9	14	0.008150	0	378
test10	15	0.008603	0	436
test11	16	0.010681	52	2106
test12	18	0.015772	53	2717
test13	20	0.014355	0	742
test14	22	0.014459	0	420
test15	24	0.025679	51	4759
test16	25	0.019769	0	244
test17	26	0.022319	0	894
test18	28	0.026074	0	1528
test19	30	0.026810	0	776
test20	30	0.025305	0	434
test21	50	0.078030	0	1714
test22	75	0.196665	0	1368
test23	100	0.399024	0	1092
test24	200	6.896993	19	107527
test25	300	98.771983	5923	634689

Table 2: Metrics Summary for CSP Solver (OR-Tools)

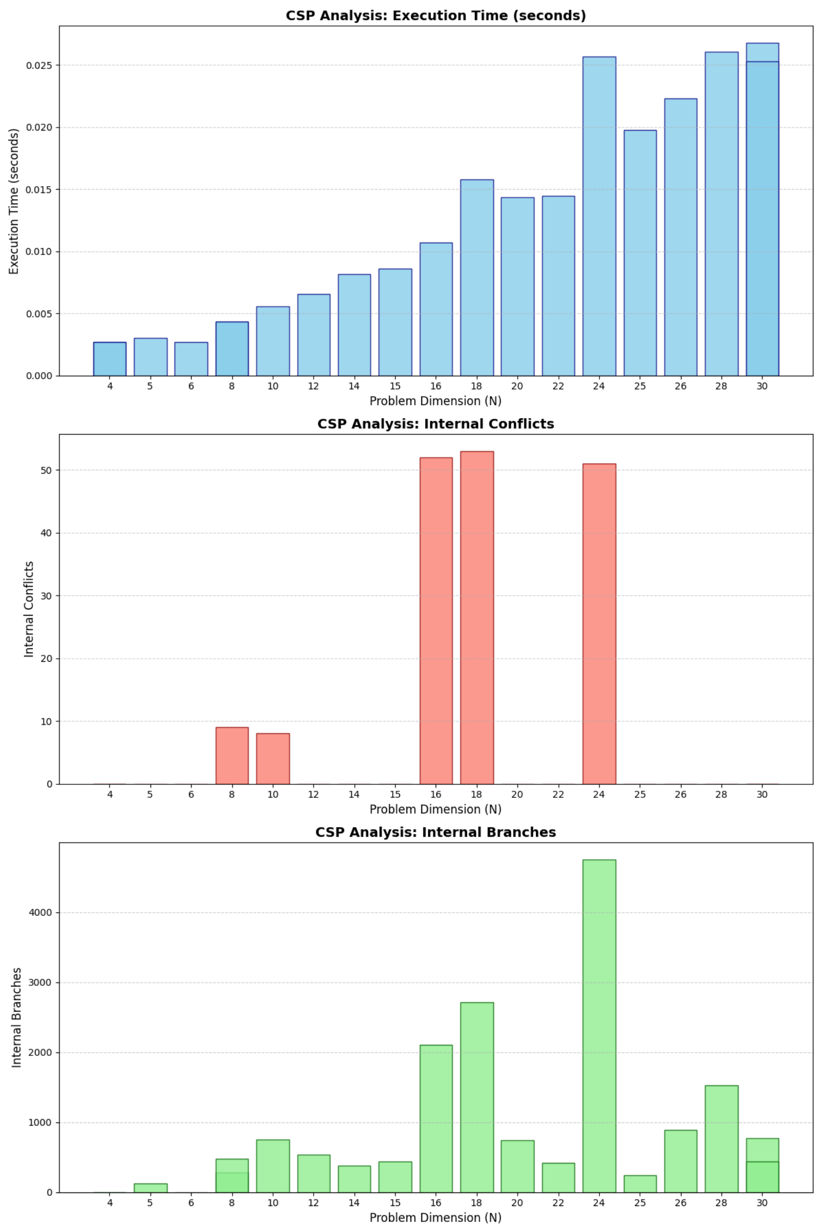

Figure 2: CSP metrics graph

5.1 Final Discussions and Key Findings

Based on the experimental data collected across both A* Search and CSP paradigms, several critical considerations emerge about the nature of the N-Queens problem:

- **Search vs. Inference Paradigms:** The most significant finding is the fundamental difference in how these algorithms approach the solution space. While A* relies on an *exploratory* process, attempting to build a path toward the goal by evaluating heuristic estimates, the CSP solver utilizes *logical inference*. By applying global constraints, the CSP solver prunes impossible configurations before they are even visited, proving that for highly constrained problems, inference is shown to be superior to state-space search for this problem.

- **Initial State Sensitivity vs. Robustness:** A* search demonstrated high sensitivity to the initial condition, where performance was heavily dictated by the "luck" of the initial queen distribution. In contrast, the CSP solver proved to be exceptionally robust: since it does not rely on a starting state but rather on the logical reduction of domains, it provides consistent and predictable execution times, which is an essential requirement for real-world production systems.
- **Reliability and Validation:** Despite the radical difference in speed, both approaches maintained a 100% success rate in finding valid solutions (evidenced by the zero-conflict heuristic check). This confirms that the choice between search and constraint programming is not one of solution quality, but strictly one of computational efficiency and scalability.

6 Conclusion

This study provided a comprehensive comparison between state-space search and constraint satisfaction paradigms using the N-Queens problem as a benchmark. The results clearly demonstrate that while the A* algorithm is a valuable tool for understanding heuristic search and local optimization, its practical application is limited by the exponential growth of nodes and the computational overhead of state evaluation in large dimensions.

In contrast, the CSP approach, powered by the OR-Tools CP-SAT solver, proved to be the superior methodology for this class of problem. By transforming the search task into one of logical inference and domain reduction, the CSP model achieved industrial-grade scalability, solving in milliseconds what took the A* algorithm nearly a minute. Ultimately, this project highlights that the choice of an appropriate paradigm is as critical as the algorithm itself; for highly constrained combinatorial problems, leveraging specialized constraint propagation engines offers a level of efficiency and robustness that traditional search methods cannot match.