Artificial Intelligence Homework: A* Search and CSP for the n-Queens Problem

Name: Giorgia Saolini Student ID: 1933076

1. Introduction

This homework addresses the classical n-Queens problem using two Artificial Intelligence techniques: A* search and Constraint Satisfaction Problems (CSP). The goal is to design and evaluate a complete experimental pipeline, from problem modeling to a comparative analysis of the two approaches, highlighting how different AI paradigms behave on the same combinatorial problem as its size increases.

The n-Queens problem consists of placing n queens on an n × n chessboard so that no two queens attack each other, meaning they cannot share the same row, column, or diagonal. Although simple to state, the problem has a rapidly growing search space, making it a standard benchmark for testing AI techniques.

The problem is first modeled as a state-space search suitable for A*. A state is represented as a partial assignment of queens to rows, encoded as a tuple where each value indicates the column of a queen. This representation guarantees row uniqueness and allows incremental construction of solutions. The initial state is empty, actions consist of placing a queen in the next row, and invalid placements are pruned by checking column and diagonal conflicts. A goal state contains n non-attacking queens, with unit cost actions.

A* is implemented with duplicate elimination and no node reopening. An admissible heuristic is used, based on the number of conflicts among already placed queens. While simple, this heuristic is sufficient to demonstrate how A* performance depends on heuristic guidance and suffers from scalability issues due to memory and search space growth.

The second approach formulates the problem as a CSP. Each row is a variable with domain {0, ..., n−1}, and constraints enforce column uniqueness and diagonal non-attacks. The CSP is solved using the CP-SAT solver from Google OR-Tools, which exploits constraint propagation and efficient search strategies.

Finally, both approaches are evaluated on increasing values of n. For A*, metrics such as runtime, expanded nodes, and frontier size are collected, while for CSP, runtime and solver statistics are recorded. The results show that A* struggles to scale due to explicit search, whereas CSP performs significantly better thanks to strong constraint propagation. Overall, the homework illustrates the impact of different modeling and solving paradigms on performance and emphasizes the importance of choosing the appropriate AI technique for a given problem.

2. Task 1 – Problem Description

2.1 Problem Definition

The n-Queens problem consists of placing n queens on an n × n chessboard so that no two queens attack each other. According to the rules of chess, this requires satisfying three constraints: no two queens can share the same row, the same column, or the same diagonal. Despite its simple definition, the interaction of these constraints produces a highly combinatorial search space that grows exponentially with n, making brute-force approaches infeasible for moderate board sizes.

Because of its clear goal, strong constraints, and scalability, the n-Queens problem is widely used as a benchmark in Artificial Intelligence. It can be naturally formulated both as a state-space search problem and as a Constraint Satisfaction Problem, making it well suited for comparing different AI

paradigms. In this homework, it provides the common basis for evaluating A* search and CSP-based solving.

2.2 State Representation

A state is represented as a tuple of integers, where the index corresponds to a row and the value to the column in which the queen is placed. For example, the state (1, 3, 0) represents three queens placed in rows 0, 1, and 2 at the specified columns. This representation implicitly enforces the row constraint, is compact, and supports incremental construction of solutions.

Using tuples ensures that states are immutable and hashable, enabling efficient duplicate detection in A*. Moreover, partial assignments allow early detection of conflicts, improving pruning and overall search efficiency.

2.3 Initial State

The initial state is the empty tuple (), representing a chessboard with no queens placed. It corresponds to the root of the search tree and allows the algorithm to build solutions incrementally without any initial constraints. This choice simplifies the implementation, as the initial state is always valid and requires no special handling.

2.4 Actions

An action consists of placing a queen in the next available row. Given a state of length k, the queen is placed in row k by choosing any column from 0 to n−1, producing a new state by appending that column to the tuple.

Actions that result in column or diagonal conflicts with existing queens are immediately discarded. This early pruning significantly reduces the search space. Placing queens row by row is a deliberate modeling choice that reduces the branching factor and aligns naturally with the CSP formulation.

2.5 Goal Test

A state is a goal if it contains exactly n queens and satisfies all constraints. Since invalid states are pruned during generation, the goal test effectively reduces to checking whether the length of the state tuple is n. Reaching a goal state terminates the search and yields a complete, valid configuration.

2.6 Cost Function

Each action has a uniform cost of 1, corresponding to placing one queen. Therefore, the path cost of a state equals the number of queens placed so far. All goal states have the same total cost n, reflecting the fact that the problem is not about optimization but about finding any valid solution. This uniform cost simplifies A* and preserves heuristic admissibility.

3. Task 2.1 – A* Search Implementation

3.1 Algorithm Description

In this section, we describe the implementation of the A* search algorithm used to solve the n-Queens problem. The algorithm follows exactly the version presented in the course slides, as required by the assignment. In particular, duplicate elimination is enforced and node reopening is

not allowed, choices that influence both the algorithm's behavior and the interpretation of the results.

A* is a best-first search algorithm that selects, at each step, the node with minimum estimated total cost, defined as f(n) = g(n) + h(n), where g(n) is the path cost from the initial state and h(n) is a heuristic estimate of the remaining cost to reach a goal. While A* is complete and optimal under certain conditions, this homework intentionally adopts a simplified version without node reopening, in line with the assignment specifications.

The algorithm maintains two main data structures: the frontier and the explored set. The frontier contains generated but unexpanded nodes and is implemented as a priority queue ordered by increasing f-values. The explored set stores states that have already been expanded and is used to prevent duplicate expansions.

Initially, the frontier contains only the initial state, with g = 0 and h computed by the heuristic, while the explored set is empty. At each iteration, the node with the smallest f-value is extracted. If it satisfies the goal test, the algorithm terminates and reconstructs the solution path. Otherwise, the node's state is added to the explored set and expanded by generating successor states. Successors whose states already appear in the explored set are discarded, and no checks for better paths are performed due to the absence of node reopening.

This simplified version of A* is sufficient for the n-Queens problem, which is structured as a tree search with incremental placements, making duplicate states relatively uncommon. Finally, the implementation is modular: A* is independent of the specific problem and interacts with it through a well-defined interface. This design improves clarity, maintainability, and reusability for other search problems.

3.2 Heuristic Function

The heuristic function is a key component of the A* algorithm, as it guides the search toward more promising states. In this homework, we use a simple heuristic specifically designed for the n-Queens problem, which estimates the remaining cost by counting conflicts among the queens already placed.

Given a partial state with k queens, the heuristic counts the number of conflicting pairs. Two types of conflicts are considered: queens placed in the same column and queens placed on the same diagonal. For each pair of queens, a conflict is detected if they share a column or if the difference between their column indices equals the difference between their row indices. Each detected conflict contributes one unit to the heuristic value.

This heuristic is admissible because it never overestimates the cost to reach a goal state. Each conflict represents a constraint violation that must be resolved by future actions, and since each action has unit cost, the heuristic provides a lower bound on the remaining cost. However, the heuristic is relatively weak, as it only accounts for existing conflicts and does not anticipate future ones. This simplicity is intentional, as the aim is to provide a clear and correct A* implementation rather than an optimized heuristic.

From a computational perspective, the heuristic is inexpensive to compute, requiring $O(k^2)$ time for k placed queens. The heuristic value is computed once per generated node and stored, allowing efficient evaluation of the f = g + h function during node comparison in the priority queue.

3.3 Data Structures

The A* implementation uses a small set of well-chosen data structures to ensure correctness and efficiency. Each search node stores the state of the board, represented as a tuple of integers, a reference to its parent node for solution reconstruction, the path cost g, the heuristic value h, and the evaluation function f = g + h.
Nodes are ordered in the frontier according to their f-values, with ties broken arbitrarily by the priority queue. The frontier itself is implemented using Python's built-in heapq module, which provides efficient logarithmic-time insertion and removal.
The explored set is implemented as a Python set containing the states that have already been expanded. Since states are immutable and hashable tuples, membership checks can be performed in constant time, enabling efficient duplicate elimination.
The use of standard Python data structures complies with the assignment guidelines and ensures a clear, readable implementation while remaining faithful to the theoretical A* algorithm presented in the course.

3.4 Metrics Collected

To evaluate the performance of the A* implementation and compare it with the CSP-based approach, several metrics are collected during execution. These metrics capture both time and space aspects of the search process.
The first metric is the total running time, measured as wall-clock time from the start of the algorithm until a solution is found. This reflects the overall efficiency of the implementation, including heuristic evaluation, node expansion, and data structure operations.
The second metric is the number of expanded nodes, defined as nodes removed from the frontier and expanded. This is a standard, machine-independent measure of search effort. The third metric is the number of generated nodes, which indicates how many nodes are created and added to the frontier and provides insight into the growth of the search space.
Finally, the maximum size of the frontier is recorded as an approximation of peak memory usage, since the frontier is the main memory-consuming structure in A*. Together, these metrics offer a clear view of the efficiency and scalability of A* and enable a meaningful comparison with the CSP-based solution.

4. Task 2.2 – Constraint Satisfaction Problem (CSP)

4.1 CSP Modeling

In this task, the n-Queens problem is modeled as a Constraint Satisfaction Problem (CSP). Unlike state-space search methods such as A*, CSPs adopt a declarative approach in which the problem is defined by variables, domains, and constraints. A solution corresponds to an assignment of values to all variables that satisfies all constraints simultaneously. This paradigm is well suited to n-Queens, as the problem is naturally expressed in terms of constraints on queen positions.

Variables

The model defines one variable for each row of the chessboard, (Q0, Q1, ..., Qn−1), where Qi represents the column of the queen in row i. This choice implicitly enforces the row constraint and aligns with the state representation used in the A* approach, allowing solutions to be represented as sequences of length n and facilitating comparison between the two methods.

Domains
Each variable Qi has domain {0, 1, ..., n−1}, corresponding to the possible columns of the board. Assigning a value to a variable places a queen at the corresponding position. The domains are finite, uniform, and relatively small, making the problem suitable for finite-domain solvers such as CP-SAT.

Constraints
The constraints encode the rules of the n-Queens problem. Column constraints require that no two variables take the same value, i.e., $Q_i \neq Q_j$ for all $i \neq j$, effectively enforcing an all-different condition. Diagonal constraints ensure that no two queens share a diagonal, imposing $|Q_i - Q_j| \neq |i - j|$ for every pair of distinct rows.
Although the number of constraints grows quadratically with n, CSP solvers efficiently handle this through constraint propagation and pruning. Overall, the CSP formulation is compact and expressive, focusing on what constitutes a valid solution rather than how to search for it, and it scales well for the n-Queens problem.

4.2 Solver Integration
To solve the CSP, we use the CP-SAT solver from Google OR-Tools, a state-of-the-art solver that combines constraint programming and SAT techniques. It supports integer variables and complex constraints, and relies on propagation, backtracking search, and conflict-driven learning.
As required by the assignment, the solver is fully integrated into the codebase and used programmatically. The solving process consists of three main phases: model construction, solver invocation, and solution parsing.

Model Construction
For each value of n, the CSP model is built dynamically in code by declaring variables, defining their domains, and adding all required constraints. This approach ensures scalability, consistency with the problem definition, and reproducibility of the experiments.

Solver Invocation
Once the model is created, the CP-SAT solver is called through the OR-Tools API. The solver autonomously explores the search space using constraint propagation and backtracking. This clearly separates problem modeling from the search process, in contrast to A*, where all search mechanisms must be explicitly implemented.

Solution Parsing and Conversion
When a solution is found, the assignment to variables $Q_0,\ldots,Q_{n-1}$ is extracted and converted into a tuple representing queen positions. This representation is identical to the one used by A*, enabling direct comparison, validation, and analysis of solutions produced by the two approaches.

4.3 Metrics Collected
To evaluate the CSP approach and compare it with A*, several metrics are collected during solver execution to assess efficiency and search complexity.

The first metric is runtime, measured as the total wall-clock time to build the model, invoke the solver, and retrieve a solution. This allows direct comparison with A*.
Two solver-specific metrics are also recorded: the number of branches and the number of conflicts. Branches correspond to decision points where the solver assigns a value or splits a domain, indicating the size of the explored search space. Conflicts count how often the solver encounters inconsistent assignments that violate constraints, triggering backtracking and learning. A high number of conflicts indicates a challenging search, while a low number reflects effective constraint propagation.
Unlike A*, CSP solvers do not maintain an explicit frontier, so memory usage is not measured the same way. These metrics together—runtime, branches, and conflicts—provide a clear picture of how the CSP approach scales with n and illustrate its strengths compared to search-based methods like A*.

5. Task 3 – Experimental Results

5.1 Experimental Setup

This section presents the experimental evaluation of A* search and CSP on the n-Queens problem. The goal is to compare their behavior as n increases, highlighting differences in scalability, efficiency, and resource use.
Experiments were conducted on problem instances with n = 4, 6, 8, 10, 12. Small n values are easy to solve, while larger n increase complexity and challenge search-based approaches, allowing observation of each algorithm's response to the growing search space.
For each n, both A* and CSP solved the same instance. A* used duplicate elimination, no node reopening, and the conflict-count heuristic, while CSP was solved automatically with the CP-SAT solver.
Each run was independent, and performance metrics were recorded. For A*: runtime, expanded nodes, generated nodes, and maximum frontier size; for CSP: runtime, solver branches, and solver conflicts. These metrics capture time and search complexity, memory usage, and solver behavior.
All experiments were performed in the same environment to ensure fairness. While absolute runtimes may vary by hardware, the observed trends reliably illustrate the comparative performance of the two approaches.

5.2 Results Overview

The experimental results highlight clear differences between A* search and CSP on the n-Queens problem. Overall, A* exhibits near-exponential growth in runtime and memory as n increases, while CSP scales more gracefully and remains efficient for larger instances.
For n = 4 and 6, both algorithms solve the problem quickly. A* explores only a few states with a small frontier, while CSP requires few branches and conflicts. At this scale, runtime differences are negligible.
For n = 8, A* begins to generate and expand many more nodes, and frontier size grows, increasing memory usage. CSP remains efficient, with more branches and conflicts but still benefiting from constraint propagation to prune the search space.
At n = 10, A* runtime and memory usage increase significantly, reflecting the combinatorial explosion. CSP continues to solve the problem efficiently, handling more branches and conflicts without dramatic resource increases.

For n = 12, A* struggles, with sharp rises in runtime and memory, sometimes approaching practical limits. CSP still performs well, as constraint propagation and learned constraints allow it to prune most invalid assignments.
In summary, while both methods solve small instances efficiently, A* scales poorly, whereas CSP remains practical and efficient as problem size grows.

5.3 Discussion
The experimental results confirm the theoretical expectations regarding A* and CSP. The differences in performance stem from how each approach explores and prunes the search space.
A* is a general-purpose heuristic search that explicitly constructs a search tree. Even with pruning and an admissible heuristic, it must consider many partial configurations. Each added row introduces up to n column choices, so the branching factor remains high. The conflict-count heuristic guides the search but offers limited insight into future conflicts, leading A* to explore many locally promising but ultimately invalid states. Memory usage is also significant, as both the frontier and explored set grow with the number of generated nodes, creating potential bottlenecks.
In contrast, CSP does not enumerate partial states explicitly. It uses a declarative model and constraint propagation to eliminate inconsistent values before they form complete assignments. Column and diagonal constraints prune large portions of the search space early. Solver-specific metrics, such as branches and conflicts, grow much more slowly with n than A*'s expanded nodes, reflecting the efficiency of learning and propagation.
CSP also benefits from domain-specific reasoning unavailable in generic A*, including global constraint propagation, backjumping, and clause learning. These techniques exploit the strong interdependence of constraints in n-Queens, giving CSP a substantial advantage over search-based methods.
From an AI perspective, these results illustrate the importance of problem representation and algorithm choice. While A* is flexible and effective in domains with informative heuristics and loose constraints, CSP excels in highly constrained combinatorial problems by leveraging constraint-based reasoning. The comparison highlights the strengths and limitations of each approach, emphasizing the need to match the solving paradigm to the problem structure.

6. Conclusion
In this homework, we implemented and compared two AI techniques—A* search and CSP—to solve the n-Queens problem. The goal was not only to find solutions, but also to design a complete AI pipeline including problem modeling, algorithm implementation, solver integration, and experimental evaluation, providing both theoretical and practical insights.
The n-Queens problem, despite its simple formulation, has a rapidly growing search space and strong variable interdependencies, making it ideal for comparing search-based and constraint-based approaches. A* models the problem as a state-space search, with partial queen placements forming explicit states and incremental actions building solutions. Implementing A* highlighted the role of the evaluation function, heuristic design, and frontier management. However, even with pruning and duplicate elimination, A* suffers from exponential growth in runtime and memory for larger n, as the heuristic only partially guides the search.
In contrast, the CSP approach models the problem declaratively with variables, domains, and constraints, delegating search complexity to a specialized solver. The CP-SAT solver from

OR-Tools uses constraint propagation, backtracking, and conflict-driven learning to prune the search space, avoiding many configurations that A* must explore. Experiments showed that while both approaches perform similarly on small instances, CSP scales far better for larger n, with lower runtime and resource usage.

This comparison highlights key lessons in AI. General-purpose algorithms like A* are flexible but may struggle with highly constrained problems, whereas CSP solvers exploit structure and constraints for superior efficiency. Effective modeling is crucial: by focusing on what constitutes a valid solution rather than how to construct it, CSP allows powerful reasoning techniques that are difficult to implement manually. Integration of external solvers, automated model generation, and programmatic solution parsing are important practical skills, ensuring reproducibility and fair comparison.

Experimental evaluation provided deeper insight into algorithm behavior. Metrics such as expanded nodes, frontier size, branches, and conflicts revealed not just which method performs better, but why. This reinforces the value of empirical analysis alongside theoretical study, showing that practical efficiency depends on problem structure, solver capabilities, and implementation details.

Overall, the homework demonstrates how different AI paradigms can be applied and compared systematically. A* offers transparency and intuitive reasoning, while CSP provides superior scalability for highly constrained combinatorial problems like n-Queens. The project underscores the importance of careful modeling, modular design, solver integration, and experimental analysis, showing that effective AI solutions often emerge from combining multiple approaches rather than relying on a single algorithm.