Artificial Intelligence Homework 2026

Flavio Visaggio 2260384

09/01/2026

1 Introduction

The goal of this project is to design and implement a complete AI pipeline: modeling a specific problem, applying different techniques to solve it, and analyzing the performance through experimental results.

Among the available options, the **N-Queens problem** was chosen. This is a classical combinatorial puzzle that involves placing N chess queens on an $N \times N$ chessboard such that no two queens threaten each other. This problem offers an ideal benchmark for comparing search strategies due to its scalability and the nature of its constraints.

Two different AI techniques were implemented and compared to solve the problem:

- **A* Search**: A heuristic search algorithm implemented. A custom heuristic based on the number of remaining queens was designed to guide the search efficiently.
- **Constraint Satisfaction Problem (CSP)**: A solver approach based on reducing the problem to a set of variables and constraints, implemented using the `python-constraint` library.

The project was developed in Python using a modular architecture that separates the problem modeling from the solver implementations. This design choice ensures code reusability and facilitates clean testing.

The third and final task of this project was to conduct experiments by scaling the board size N from 4 to 11. The experimental results analyze specific metrics of the models and highlight the significant performance differences between the two methods as the search space grows exponentially.

2 Task 1: Problem Modeling

In this section, we describe how the N-Queens problem has been modeled as a search problem. The implementation is defined in the class `NQueensProblem`, which inherits from the abstract base class `Problem`.

2.1 Problem Definition

The N-Queens problem consists of placing N queens on an $N \times N$ chessboard so that no two queens attack each other. In chess, a queen attacks any piece in the same row, column, or diagonal.

2.2 State Representation

A crucial aspect of the implementation is the state representation. Instead of representing the full $N \times N$ grid (which would be memory inefficient), we exploit the constraint that only one queen can be placed per column.

Therefore, a state is represented as a tuple of integers:

$$S = (r_0, r_1, \ldots, r_k)$$

where r_i represents the row index of the queen placed in column i.

- The index of the element in the tuple represents the column.
- The value of the element represents the row.
- The length of the tuple k indicates how many queens have been placed so far.

For example, a state $S = (1, 3)$ means:

- A queen is at column 0, row 1.
- A queen is at column 1, row 3.

2.3 Initial State and Goal Test

- **Initial State**: An empty tuple $S_0 = ()$, representing an empty board with zero queens placed.
- **Goal Test**: A state is a goal state if it contains N queens (i.e., the length of the tuple is N) and no queen attacks another. Since our successor function only generates valid states, checking the length is sufficient.

2.4 Actions and Transition Model

The actions available in a given state S consist of placing a new queen in the first available column. The get_successors method iterates through all possible rows $r \in \{0, \ldots, N-1\}$ for the current column. A new state $S' = S + (r)$ is generated only if placing the queen at row r does not violate any constraints with the queens already placed in previous columns.

This approach ensures that we essentially prune the search tree early, as we never generate invalid partial states.

3 Task 2.1: Implementation of A*

This section details the implementation of the A* search algorithm, which was developed from scratch in the search.py module. The implementation follows the standard graph search approach with duplicate elimination, as required by the assignment specifications.

3.1 The Algorithm

The A* algorithm explores the state space by expanding nodes based on the evaluation function:

$$f(n) = g(n) + h(n)$$

where:

- $g(n)$ is the path cost from the initial state to node n. In our N-Queens model, placing a queen costs 1, so $g(n)$ is simply the number of queens placed so far (the depth of the node).
- $h(n)$ is the heuristic estimated cost from n to the goal.

As required by the assignment, the implemented A* corresponds to a graph search variant with duplicate elimination and no reopening of closed nodes. Once a state is inserted into the explored set, it is never added back to the frontier, ensuring that each reachable state is expanded at most once.

3.2 Data Structures

To ensure efficiency, the following data structures were used:

- **Frontier**: Implemented as a priority queue using Python's heapq module. This allows extracting the node with the lowest $f(n)$ in $O(\log N)$ time.
- **Explored Set**: Implemented as a hash set (set() in Python) to store the states that have already been visited. This design guarantees duplicate elimination: any state that appears in the explored set is not reinserted into the frontier, thus preventing node reopening and avoiding unnecessary re-expansions.
- **Frontier Lookup**: A dictionary frontier_states tracks states currently in the frontier to enable efficient updates if a better path to an existing state is found.

3.3 The Heuristic Function

For the A* algorithm to be effective, an appropriate heuristic $h(n)$ is required. Defining a custom heuristic in the NQueensProblem class:

$$h(n) = N - \text{number of placed queens}$$

This heuristic represents the number of rows remaining to be filled.

- **Admissibility**: Since we must place exactly one queen per row to reach a solution of size N, and each action places one queen, we need *at least* $N - k$ steps to complete a partial solution of k queens. Therefore, $h(n)$ never overestimates the true cost to reach the goal.
- **Consistency**: The cost of moving from one state to a successor is 1. The heuristic value decreases by exactly 1 at each step (since one more queen is placed).

$$h(n) \leq c(n, a, n') + h(n')$$

Takes to:

$$(N - k \leq 1 + (N - (k + 1)))$$

making the heuristic consistent and guaranteeing an optimal solution.

4 Task 2.2: Implementation of CSP

For the second AI technique,the objective is to model the N-Queens problem as a Constraint Satisfaction Problem (CSP) and solve it using a generic CSP solver. This approach differs significantly from A*: instead of searching through a state space of partial assignments, we define the variables and the rules (constraints) that a valid solution must satisfy, letting the solver's engine handle the constraint propagation and variable instantiation.

4.1 Setup

The implementation relies on the python-constraint library, which provides a flexible framework for defining variables, domains, and constraints. The logic is encapsulated in the csp_solver.py module.

4.2 Problem Reduction

To reduce the N-Queens problem to a CSP, we defined the following components:

- **Variables**: We created N variables, one for each column of the chessboard: $X = \{C_0, C_1, \ldots, C_{N-1}\}$.
- **Domains**: The domain for each variable represents the possible row indices where a queen can be placed. Since the board is $N \times N$, the domain for every variable is $D = \{0, 1, \ldots, N - 1\}$.

4.3 Constraints

Two main types of constraints were applied to ensure a valid N-Queens configuration:

1. **Row Constraint**: Since each variable represents a column and its value represents a row, ensuring that no two queens share the same row is equivalent to enforcing that all variables must have distinct values. To realize it I utilized the AllDifferentConstraint() provided by the library.
2. **Diagonal Constraint**: To preventing diagonal attacks, I enforced a binary constraint between every pair of variables (C_i, C_j) (where $i < j$). Two queens at (C_i, R_i) and (C_j, R_j) are on the same diagonal if and only if the absolute difference between their rows equals the absolute difference between their columns. Therefore, the constraint is satisfied if:

   $$|R_i - R_j| \neq |C_i - C_j|$$

   This logic was implemented using a lambda function and applied to all distinct pairs of columns.

4.4 Solving Process

Once the problem is defined, the solver uses a backtracking search algorithm combined with constraint propagation (typically ensuring arc consistency) to prune values from domains that violate constraints. This allows the CSP solver to find a solution (or determine that none exists) much faster than an unguided search, especially for larger values of N.

5 Task 3: Experimental Results

This section analyzes the performance of the two implemented algorithms. The experiments were conducted by scaling the board size N from 4 to 11. The metrics collected include execution time and, for the A* algorithm, the number of expanded nodes and maximum memory usage.

The experiments were automated using a dedicated script (experiments.py) to ensure consistency.

5.1 Performance Comparison: Time

The execution time for both algorithms was measured on a standard consumer machine running Python 3.13.

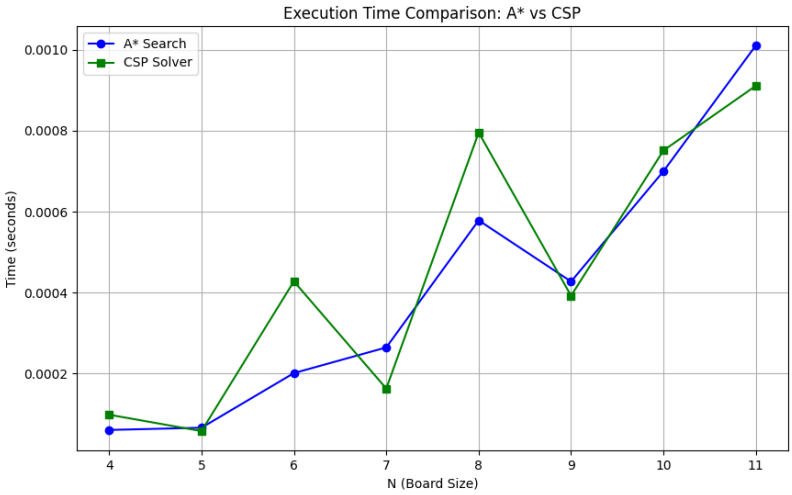

Figure 1: Execution Time Comparison: A* vs CSP (N=4 to N=11)

As shown in the collected data, both algorithms perform extremely well for small instances ($N \leq 11$).

The CSP Solver proved to be the most efficient approach, maintaining a near-constant execution time of approximately 0.0001 - 0.0009 seconds regardless of N. The overhead of creating variables and constraints is negligible, and the constraint propagation engine solves these instances instantly.

The A* Search also performed admirably. While the execution time shows a slight upward trend as N increases (reaching roughly 0.001s for $N = 11$), it

remains highly competitive. This suggests that for constrained sizes, a well-designed heuristic can make search algorithms viable.

5.2 Search Space Analysis (A*)

A critical metric for A* is the number of expanded nodes, which correlates directly to the difficulty the algorithm faces in finding the goal.

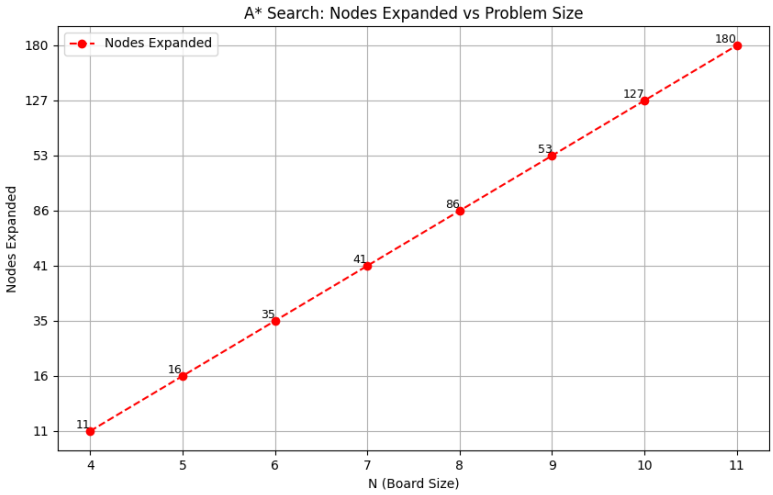

Figure 2: A* Search: Nodes Expanded vs Problem Size N

The results highlight the effectiveness of the chosen heuristic $h(n) = N - \text{placed\_queens}$. For $N = 4$, A* expanded only 11 nodes. For $N = 8$, it expanded 86 nodes, and for $N = 11$, it expanded 180 nodes.

Considering that the total state space for N-Queens grows factorially ($N!$), the fact that the expanded nodes grow almost linearly (as shown in Figure 2) implies that the heuristic is highly informative, drastically pruning the search tree.

5.3 Conclusion of Experiments

The experiments demonstrate that while A* is a valid strategy when equipped with a strong heuristic, the reduction to CSP is superior for this specific class of problems. The ability of the CSP solver to prune inconsistent values from domains before even attempting assignments makes it more robust for scaling to larger N.

6 How to Run

This section outlines the steps required to set up the environment and execute the project code to reproduce the experimental results. The software is developed in Python 3.

6.1 Environment Setup

The implementation utilizes the standard Python library along with an external dependency for the Constraint Satisfaction Problem solver. The required packages are listed in the requirements.txt file.

To install the dependencies, the following command must be executed in the terminal:

```
pip install -r requirements.txt
```

6.2 Execution

Two primary scripts are provided to perform the tasks required by the assignment:

- `main.py`: This script executes a single instance of the N-Queens problem (default size $N = 8$). It runs both the A* Search and the CSP Solver, displaying the solution, execution time, and memory usage statistics.

  Command to run the single instance:

  ```
  python main.py
  ```

- `experiments.py`: This script automates the experimental phase described in Section 3. It iterates through board sizes from $N = 4$ to $N = 11$, collecting performance metrics for both algorithms. The results are printed to the standard output and saved to a CSV file named nqueens_experiments.csv.

  Command to run the benchmark suite:

  ```
  python experiments.py
  ```

7 Conclusion

In this project, an end-to-end AI pipeline was developed to solve the N-Queens problem. The work involved modeling the problem domain, implementing two distinct search strategies, and conducting a comparative analysis based on experimental data.

The comparison between the A* Search algorithm and the Constraint Satisfaction Problem (CSP) solver highlighted significant differences in efficiency and scalability.

The A* algorithm, implemented with a domain-specific heuristic ($h(n) = N - \text{placed queens}$), demonstrated that an informed search can effectively solve the problem for moderate sizes ($N \leq 11$) with a manageable number of expanded nodes. However, the exponential growth of the search space remains a limiting factor for larger instances.

Conversely, the CSP approach proved to be superior for this specific combinatorial problem. By leveraging constraint propagation and an optimized backtracking solver, it consistently found solutions in negligible time, regardless of the board size within the tested range.

This analysis confirms that while heuristic search is a powerful general-purpose tool, problem reduction to a specialized framework like CSP is often the optimal strategy for problems defined by strict constraints.