AI Homework: Sudoku Solver Analysis

Lorenzo Gaudino 2046679

January 2026

Introduction

This report analyzes the application of Artificial Intelligence techniques to the game of Sudoku.

The project focuses on the implementation of two distinct solvers: a search-based agent utilizing the A* algorithm, optimized with domain-specific heuristics and the *Minimum Remaining Values (MRV)* strategy, and a declarative solver based on Constraint Satisfaction (CSP) using the `python-constraint` library. The following sections detail the problem modeling, the algorithmic implementation, and a comparative experimental analysis of the performance and scalability of both approaches across datasets of varying complexity.

The theoretical modeling adopted in this project is based on Peter Norvig's essay *Solving Every Sudoku Puzzle*[1] and the foundational concepts presented in the textbook *Artificial Intelligence: A Modern Approach* by Russell and Norvig.

1 Task 1: Problem

1.1 Problem Selection and Definition

For this project, I selected **Sudoku**, a well-known combinatorial number-placement puzzle. While Sudoku is popularly known as a recreational game, in the context of Artificial Intelligence it serves as an ideal benchmark for studying **Constraint Satisfaction Problems (CSPs)** and search algorithms.

To formally define the problem, I adopted the terminology proposed by Peter Norvig, which provides a precise mathematical framework. The game is played on a grid of 81 **squares**, where each square must eventually contain a digit $d \in \{1, \dots, 9\}$. These squares are grouped into 27 **units** (9 rows, 9 columns, and 9 distinct 3×3 boxes).

The fundamental rule of the puzzle is the *Uniqueness Constraint*. A valid solution requires that every **unit** (row, column, and 3×3 box) contains a permutation of the digits 1 through 9 without repetition.

From a Constraint Satisfaction perspective, this implies that a value assigned to a square must not conflict with any of its **peers**—defined as the set of 20 other squares that share the same row, column, or box.

1.2 Complexity and Motivation

My choice of Sudoku was driven by personal familiarity with the game mechanics and a specific academic interest: I aimed to apply the **A* search algorithm** to a domain that is not a traditional pathfinding problem (like maze solving or navigation). Applying A* to a logical puzzle presents unique challenges in heuristic design, as physical distance is replaced by "distance to goal state" in a combinatorial space.

The problem's complexity further justifies the use of AI. While the rules are simple, the state space is huge. A naive approach considering all possible fillings would involve 9^{81} combinations. This magnitude renders simple brute-force approaches infeasible, necessitating the use of intelligent techniques such as **Informed Search (A*)** and **Constraint Propagation**.

[1] `https://norvig.com/CQ/sudoku.html`

1.3 Implementation Strategy

I implemented the solver in **Python**, using an Object-Oriented design to ensure modularity between the problem logic and the solving algorithms. I structured the system around three key components:

1.3.1 State Representation

I encapsulated the core logic in the `Sudoku` class. Rather than using a 2D matrix, I chose to flatten the grid into an array of 81 integers (where 0 represents an empty cell) to optimize memory usage and copying operations. A feature of this class is the `valid_values(index)` method, which computes the domain of a specific variable by inspecting its peers. This effectively implements the notion of *candidates*, which is essential for both the heuristic evaluation in A* and the domain reduction in the CSP solver.

1.3.2 Modeling as a Search Problem (A*)

To apply the A* algorithm, I translated the CSP definition into a sequential search problem:

- **States:** Any partial or complete assignment of the grid.
- **Actions:** I defined an action as a tuple $(index, value)$, representing the decision to place a specific digit in an empty square.
- **Transition Model:** I adopted a functional approach where applying an action returns a *new* state instance, preserving the parent state. This allows the search algorithm to backtrack naturally.
- **Cost & Goal:** The path cost is uniform (1 per step), and the goal test verifies that no zeros remain and no constraints are violated.

1.3.3 Modeling as a CSP

For the alternative AI technique, I modeled the problem declaratively using the `python-constraint` library. The motivation for this choice lies in the nature of Sudoku itself: it is defined purely by constraints (AllDifferent on rows, columns, and boxes) rather than by a path to a destination. This structure makes it an ideal candidate for Constraint Programming, allowing the problem to be solved via domain reduction and propagation rather than explicit state-space search.

In this model, the 81 squares are treated as variables with domains $\{1 \dots 9\}$. Instead of defining "moves", I simply imposed 27 `AllDifferent` constraints corresponding to the puzzle's units, allowing generalized arc consistency algorithms to solve the grid.

2 Task 2.1: Implementation of A*

The core of the intelligent search agent is implemented in the `AStarSolver` class, which orchestrates the exploration of the state space using the A* algorithm. The implementation relies on three main components: the node data structure, the heuristic evaluation, and a successor generation strategy.

2.1 Data Structures

To manage the search effectively, I defined a wrapper class `Node`. Each node encapsulates:

- **State:** The current `Sudoku` grid.
- **Costs:** The path cost $g(n)$ (depth) and the heuristic estimate $h(n)$ (estimated cost to goal).
- **Evaluation Function:** The value $f(n) = g(n) + h(n)$, used to prioritize expansion.

- **Parent Pointer:** A reference to the predecessor node, allowing the reconstruction of the solution path once the goal is reached.

The **Frontier** (or Open Set) is implemented using Python's `heapq` module, which provides a binary heap structure. This ensures that the retrieval of the node with the lowest $f(n)$ value is always an $O(\log N)$ operation. To handle cycle detection and redundant paths, an **Explored Set** (Closed Set) stores the hash of visited grid configurations, leveraging Python's efficient set lookups ($O(1)$ average time complexity).

2.2 Heuristic Function

For the heuristic $h(n)$, I implemented a domain-based approach named `h_domain_sum`. Instead of a simple zero-heuristic (which would reduce A* to Uniform Cost Search), this function calculates the sum of the cardinalities of the domains of all empty cells:

$$h(n) = \sum_{c \in \text{empty cells}} |D_c|$$

where D_c is the set of valid values for cell c given its peers. This heuristic effectively guides the search towards states where cells have fewer remaining possibilities, penalizing loose constraints.

2.3 Action Strategies

The efficiency of the search agent depends on the `get_actions` method, which determines which variable (cell) to instantiate next. To analyze the impact of variable ordering on the branching factor, I implemented three distinct strategies:

1. **First Available (Naive Baseline):** This strategy deterministically selects the first empty cell encountered in row-major order (scanning from top-left to bottom-right). It serves as a control baseline, representing a linear approach without domain knowledge.

2. **Random Selection:** This strategy selects an empty cell uniformly at random. It introduces stochasticity to the search, serving to demonstrate the high variance and inefficiency of blind guessing.

3. **Minimum Remaining Values (MRV):** This is the optimized strategy implemented to reduce the search space. Following the "Fail-First" principle, the expansion step performs a look-ahead:

   (a) The system scans all empty cells.

   (b) It identifies the single cell c_{best} with the smallest domain size $|D_c|$.

   (c) It returns actions *only* for this specific cell.

   By prioritizing the most constrained variable, MRV drastically reduces the effective branching factor, transforming the search into a chain of forced moves whenever possible.

3 Task 2.2: Implementation of CSP Solver

The second approach utilizes the **Constraint Satisfaction Problem (CSP)** paradigm. Unlike A*, which explores a tree of states, the CSP solver operates by defining variables and reducing their domains based on constraints. For this implementation, I utilized the `python-constraint` library.

3.1 Variables and Domains

The problem is modeled with 81 variables, corresponding to the indices $0 \dots 80$ of the flattened grid. To optimize the solving process before propagation begins, I applied a pre-processing step on the domains:

- **Empty Cells:** Initialized with the full domain $D_i = \{1, \ldots, 9\}$.
- **Pre-filled Cells:** Initialized with a singleton domain $D_i = \{v\}$, where v is the fixed value provided in the input string.

This pre-assignment drastically prunes the search space immediately, as the solver propagates these fixed values to remove them from the domains of all peers before any backtracking occurs.

3.2 Constraints Topology

The constraints are applied using the library's `AllDifferentConstraint`, which ensures that a set of variables takes unique values. I defined the topology as follows:

1. **Row Constraints:** 9 constraints, each applied to the 9 variables forming a row.
2. **Column Constraints:** 9 constraints, each applied to the 9 variables forming a column.
3. **Box Constraints:** 9 constraints, each applied to the disjoint 3×3 subgrids.

The solver then employs an internal backtracking algorithm combined with constraint propagation to find a variable assignment that satisfies all 27 constraints simultaneously.

4 Task 3: Experimental Results

In this section, I present the empirical analysis of the implemented algorithms. The experiments were designed to evaluate performance across two dimensions: the effectiveness of different heuristics within the A* framework and the comparative efficiency between the optimized A* agent and the CSP solver.

4.1 Experimental Setup

The benchmarks were executed on three datasets of increasing difficulty:

- **Easy:** Simple puzzles with many pre-filled cells.
- **Hardest:** Complex instances requiring deep logic.
- **Top95:** A standard benchmark of 95 extremely hard puzzles.

A timeout of **5 seconds** was imposed per puzzle to verify the feasibility of the algorithms.

4.2 Running Time and Reliability

The first comparison focuses on execution time and success rate. Table 1 summarizes the average running time for each configuration.

Table 1: Average Running Time (seconds) with a timeout of 5s.

Config_Label	CSP (-, -)	A* (MRV)	A* (First)	A* (Random)
Easy	**0.0041**	0.0118	0.7228	3.5030
Hardest	**0.0174**	0.1083	2.5826	5.0000 (TO)
Top95	**1.4088**	3.4054	5.0000 (TO)	5.0000 (TO)

As shown, the **CSP solver** is superior in terms of speed.
Among the A* configurations:

- **MRV Strategy:** It is the only search strategy that remains competitive. It solves "Hardest" puzzles in $\approx 0.1s$, which is acceptable, though slower than CSP.
- **First / Random:** These strategies exhibit bad performance on complex datasets. On the "Top95" benchmark, they failed to solve any puzzle within the 5-second limit, confirming that blind search is infeasible for Sudoku.

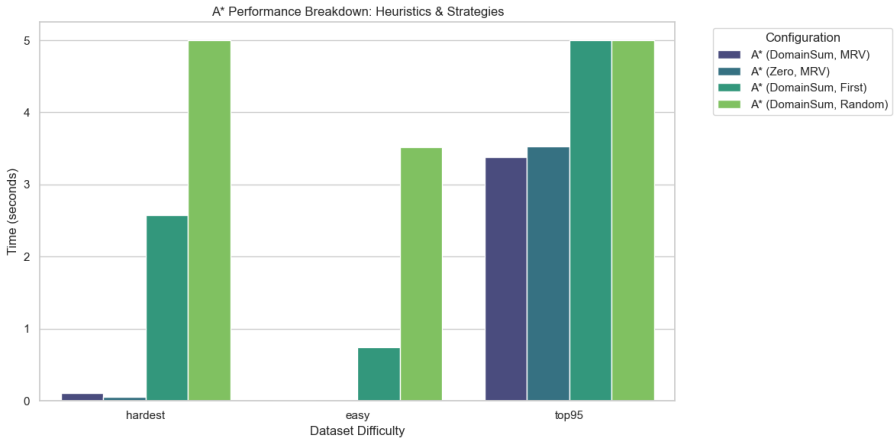

Figure 1: Comparison of execution time of A*.

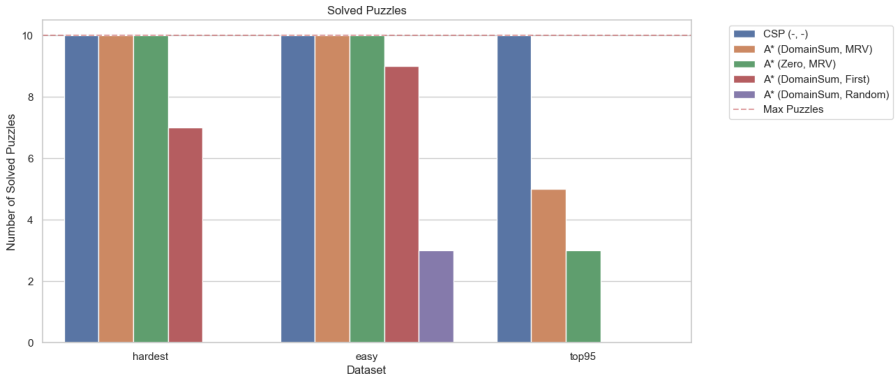

Figure 2: Success Rate per Dataset. While CSP solves 100% of instances, naive A* strategies (First, Random) fail completely on hard datasets within the timeout.

4.3 Search Efficiency and Memory

To understand why A* with MRV performs better, we analyze the number of nodes expanded and kept in memory.

As illustrated in Figure 3, the **MRV heuristic** drastically reduces the size of the search tree. On average, MRV expands 3-4 times fewer nodes than the "First Available" strategy even on easy puzzles. This reduction is exponential on harder puzzles, explaining why "First" and "Random" hit the timeout and memory limits so quickly.

4.4 Branching Factor Analysis

Finally, we examine the average Branching Factor (b^*). An ideal solver would have $b^* \approx 1$ (deterministic path).

Figure 4 highlights the core advantage of the MRV strategy. By always selecting the most constrained variable, A* (MRV) maintains a branching factor of approximately **1.1**, meaning it rarely has to guess.

5 How to run

This section provides instructions on how to set up the execution environment and reproduce the experimental results presented in this report.

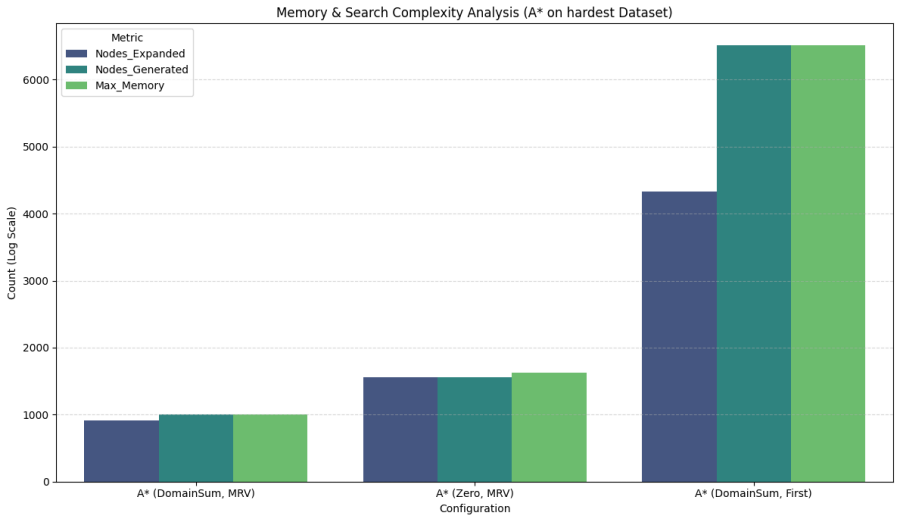

Figure 3: Comparison of Nodes Expanded and Memory Usage on the 'hardest' dataset. MRV expands significantly fewer nodes than First.

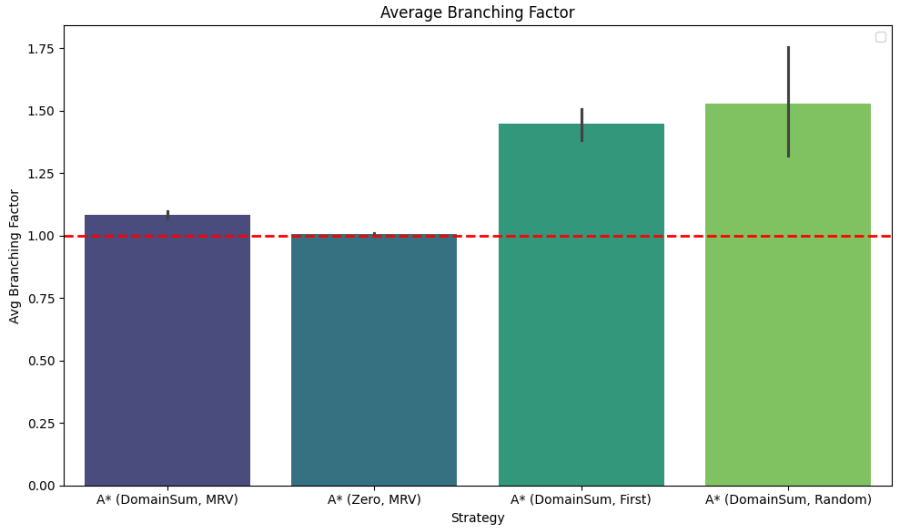

Figure 4: Average Branching Factor. The MRV strategy maintains a branching factor close to 1.0.

5.1 Environment Setup

The project is implemented in **Python 3** (tested on version 3.11). To ensure reproducibility and isolate dependencies from system packages, I developed and executed the code within a virtual environment.

1. **Virtual Environment Setup:** To replicate the environment, create and activate a virtual environment:

   ```
   python3 -m venv venv
   source venv/bin/activate  # On macOS/Linux
   # .\venv\Scripts\activate # On Windows
   ```

2. **Dependencies Installation:** All necessary libraries (including `python-constraint` for the CSP solver and data analysis tools) are listed in the `requirements.txt` file. Install them using:

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

5.2 Running the Solvers

The two solvers can be executed individually from the command line to solve a specific Sudoku instance (provided as a string of 81 characters). If no input string is provided, both scripts default to a pre-configured "Hard" instance for demonstration purposes.

A* Solver To run the A* agent (which uses the MRV strategy by default):

```
python src/AStarSolver.py "INPUT_STRING"
```

Example (runs with custom input):

```
python src/AStarSolver.py "003020600900305001001806400..."
```

Example (runs with default hard instance):

```
python src/AStarSolver.py
```

CSP Solver To run the Constraint Satisfaction solver:

```
python src/CSPSolver.py "INPUT_STRING"
```

Similarly, executing the script without arguments will solve the default instance:

```
python src/CSPSolver.py
```

5.3 Reproducing Experimental Results

Two tools are provided to replicate the performance analysis:

- **Quick Benchmark:** The script `src/benchmark.py` runs a comparative test on a predefined set of instances (Easy, Medium, Hard) and prints a tabular summary of execution times and nodes expanded directly to the console.

  ```
  python src/benchmark.py
  ```

- **Full Analysis (Notebook):** The detailed plots and extensive comparisons (Task 3) included in this report were generated using the Jupyter Notebook `Benchmark.ipynb`.