Homework AI

Sudoku Solver with A* and SAT

Antonio Lissa Lattanzio - 2154208

December 12, 2025

1 Introduction

This project compares two fundamentally different approaches to solving Sudoku: a classical heuristic search algorithm (**A***) and a constraint-based **SAT** encoding solved with a modern SAT solver. We implemented both methods from scratch, designed an experimental framework to evaluate them under controlled conditions, and analyzed their performance across puzzles with different numbers of given clues.
The pipeline includes:

- A generator of Sudoku instances with an exact number of clues.
- A complete A* solver using a state-space search over Sudoku grids.
- A SAT encoding of Sudoku with all required logical constraints.
- A unified tester that measures execution time, number of steps (A*), number of expanded/generated nodes (A*), max frontier size (A*), number of clauses/variables (SAT), clause to variable ratio (SAT) and number of unsolved puzzles for each.

The experiments show a strong asymmetry: **A*** becomes extremely slow for low-clue puzzles, while **SAT** remains fast but its formula size grows rapidly. The comparison highlights the structural differences between search-based and logic-based solving.

2 Problem

We chose the Sudoku solving problem: given a partially filled 9×9 grid, the goal is to complete it so that each row, column, and 3×3 block contains the digits 1–9 exactly once.
High-level implementation overview:

- **Puzzle representation**: a sudoku class for storing the grid, generating successor states, goal checking and utility functions.
- **A***: treated Sudoku as a search problem, where each action fills an empty cell with a legal number.
- **SAT**: encoded Sudoku as a Boolean satisfiability formula using constraints.

- **Tester**: automatically generates puzzles, solves them with both approaches, and records performance metrics and plots.

3 A* SOLUTION

The A* search algorithm is implemented in a **modular** way so it can be reused for different problems, with Sudoku being one specific application, it also support **different heuristics**. The main structure relies on a **Node** class that tracks the current puzzle state, its parent, the path cost g, the heuristic value h, and the combined score $f = g + h$. The class also defines a comparison method (__lt__) so nodes can be ordered by f within a **priority queue**.
The search process begins by creating a **start_node** from the initial Sudoku grid and pushing it onto a min-heap–based frontier. The algorithm maintains three key tracking elements: a set of **explored states**, a frontier **priority queue**, and **metrics** such as the number of generated and expanded nodes.
At each iteration, the node with the lowest f value is extracted from the frontier. If its state satisfies the Sudoku goal condition, the search terminates and reconstructs the full solution path by following parent pointers back to the root.
If it is not a goal, the algorithm marks the state as explored and expands it by generating successors. Successors are built through **child_node**, which applies an action (such as filling a cell) to the parent state, then recalculates g, applies the chosen heuristic, and returns a new node.
For each successor, the implementation checks whether its state is **already explored** or **already present** in the frontier. If it is in the frontier with a higher path cost, the old version is **replaced**. Otherwise, the new node is pushed into the frontier. Throughout the search, the algorithm tracks **statistics** like maximum frontier size and iteration count, and it stops early if an iteration limit is exceeded (set to 500000).
The **heuristic** used for Sudoku is simple: it counts the number of empty cells in the grid. This gives the solver guidance by prioritizing states that are closer to being fully filled. A better heuristic can be easily provided since the code is built to be **modular and easy to update**.
Finally, when a goal is found, the **reconstruct_path** function returns the full sequence of states from the starting puzzle to the solved grid.
Note: the maximum number of iterations of the algorithm is set to 500000 to prevent it from getting stuck for long periods.

4 SAT SOLUTION

This method encodes a Sudoku puzzle as a Boolean satisfiability (**SAT**) problem and uses the **Glucose3** solver from the **PySAT** library to find a valid assignment. The core idea is to represent each possible value in each cell as a propositional variable and then add clauses ensuring that a completed model corresponds to a legal Sudoku solution.
The function begins by creating a SAT solver instance and defines a helper function **var(i,j,n)**, which maps a row–column–digit triple to a **unique integer variable index**.

Later we will refer to propositional variables as X_{ijn}, where (i, j) indicates the cell position in the grid and n the digit.
The SAT encoding of Sudoku adds six groups of constraints, as follows:

- **Each cell must contain at least one number:** For each cell (i, j), a single clause is added containing all possible digits. For example, for the top-left cell $(0,0)$, the clause is:

  $$[X_{001}, X_{002}, X_{003}, X_{004}, X_{005}, X_{006}, X_{007}, X_{008}, X_{009}]$$

  This ensures that the cell takes at least one value. There are $9 \times 9 = 81$ such clauses.

- **Each cell must contain at most one number:** For every cell (i, j), and every pair of distinct digits (n_1, n_2), a binary clause is added to prevent both digits from being assigned simultaneously:

  $$[\neg X_{ijn_1}, \neg X_{ijn_2}]$$

  For instance, for cell $(0,0)$, clauses like $[\neg X_{001}, \neg X_{002}], [\neg X_{001}, \neg X_{003}], \ldots$ are created. This generates $9 \cdot 9 \cdot \binom{9}{2} = 2916$ clauses.

- **Row uniqueness:** For each row i and digit n, binary clauses prevent the digit from appearing in two different columns j_1, j_2 simultaneously:

  $$[\neg X_{ij_1n}, \neg X_{ij_2n}]$$

  Example: $[\neg X_{001}, \neg X_{011}], [\neg X_{001}, \neg X_{021}], \ldots$ ensuring '1' does not repeat in the first row. Total clauses: 2916.

- **Column uniqueness:** Similarly, for each column j and digit n, binary clauses prevent the digit from appearing in two different rows i_1, i_2:

  $$[\neg X_{i_1jn}, \neg X_{i_2jn}]$$

  Example: $[\neg X_{001}, \neg X_{101}], [\neg X_{001}, \neg X_{201}], \ldots$ Total clauses: 2916.

- **Respecting pre-filled entries:** Each clue in the puzzle adds a single-literal clause enforcing the value. For example, if the cell $(0,0)$ is pre-filled with 5:

  $$[X_{005}]$$

  If a puzzle has 20 clues, this adds 20 clauses.

- **Block uniqueness:** For each 3×3 block and digit n, binary clauses ensure the digit does not appear in two different cells within the block:

  $$[\neg X_{i_1j_1n}, \neg X_{i_2j_2n}]$$

  Example for the top-left block and digit 1: $[\neg X_{001}, \neg X_{011}], [\neg X_{001}, \neg X_{021}], \ldots$ Total clauses: 2916.

As clauses are generated, they are added to the solver and recorded under their respective rule labels. If the solver finds a solution, the model is parsed to reconstruct the completed Sudoku grid. Otherwise, it reports unsatisfiable.

5 Experimental Results

The experiments are conducted on random sudokus generated by a custom function with different numbers of clues (50 tests per 20, 45, 70 clues). Note: uniqueness of solution is not guaranteed, only grid validity. The evaluation framework (`tester.py`) executes and saves results with **images** made by matplotlib and a **csv table** that summarizes the whole results.

5.1 A* Metrics

For each puzzle, the A* runner records:

- **Execution time**: wall-clock time from start to solution or termination.
- **Steps**: number of state transitions in the solution path.
- **Nodes generated**: total child nodes created.
- **Maximum frontier size**: peak size of the priority queue.
- **Solution state**: final Sudoku grid if successful; otherwise `None`.

If A* fails (e.g., max 500000 iterations reached), metrics are stored as `None`.

5.2 SAT Metrics

For SAT solver:

- **Execution time**: encoding + solving duration.
- **Clauses**: total CNF clauses.
- **Variables**: distinct propositional variables.
- **Clause-to-variable ratio**: clauses / variables.
- **Solution state**: completed grid if satisfiable; otherwise `None`.

Failed SAT attempts store `None` as well.
Below the images show the results with comments.

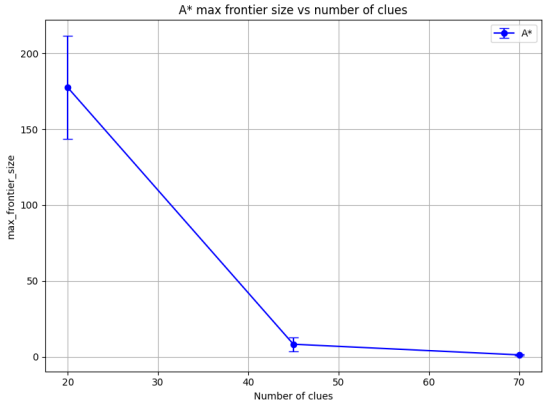

Figure 5.1: Maximum frontier size for A* search as a function of the number of clues. Each dot represents the average value across all tested puzzles, while the vertical bars indicate the standard deviation. The results show that with fewer clues (e.g., 20), the search space is larger, leading to higher maximum frontier sizes and greater variability.

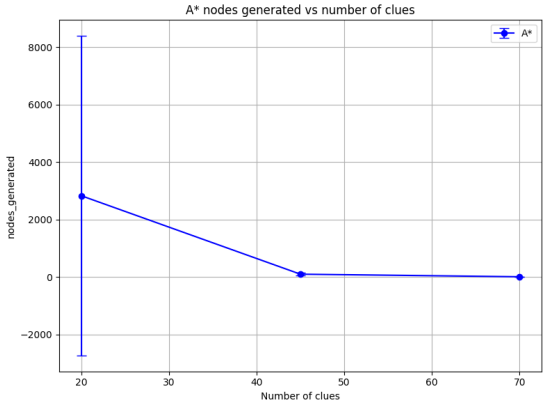

Figure 5.2: Number of nodes generated by A* for puzzles with different numbers of clues. Each dot shows the average number of nodes generated, and the vertical bars represent the standard deviation. With very few clues (e.g., 20), puzzle difficulty varies widely, so the number of nodes generated shows high variability. Note that the lower end of the bars can appear negative due to the plotting of mean minus standard deviation, but the actual node counts are always non-negative.

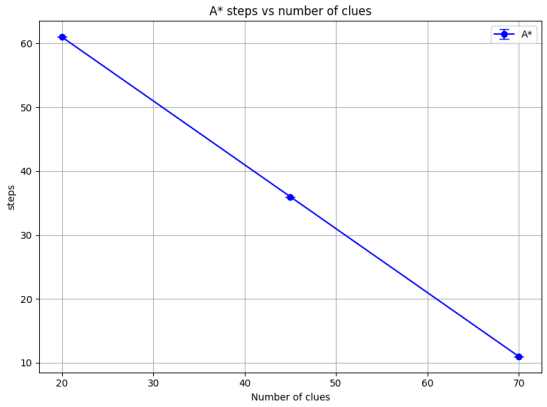

Figure 5.3: Number of steps in the solution path for A* as a function of the number of clues. Each dot represents the average number of steps across puzzles, and the vertical bars indicate the standard deviation. In this case, the standard deviation is always zero because the heuristic consistently guides the search along the same path. The number of steps decreases as the number of clues increases, reflecting the reduced search space for more constrained puzzles.

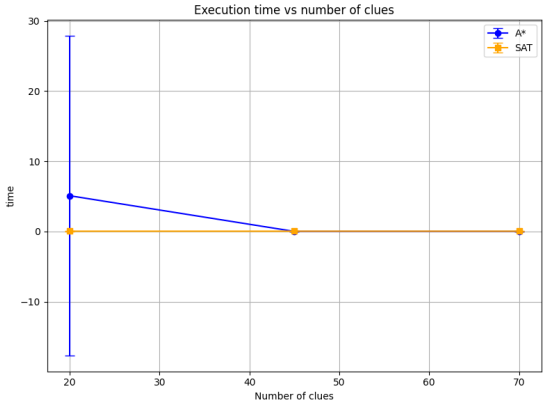

Figure 5.4: Execution time for A* and SAT solvers as a function of the number of clues. Each dot represents the average time across puzzles, and the vertical bars show the standard deviation. For very few clues (20), A* requires much more time than SAT and exhibits high variability due to the large and diverse search space. As the number of clues increases, A*'s search space becomes smaller and more constrained, leading to faster and more consistent solving times, comparable to SAT. It is worth noticing that in two cases the A* search was blocked because the maximum number of iterations (500000) was reached.

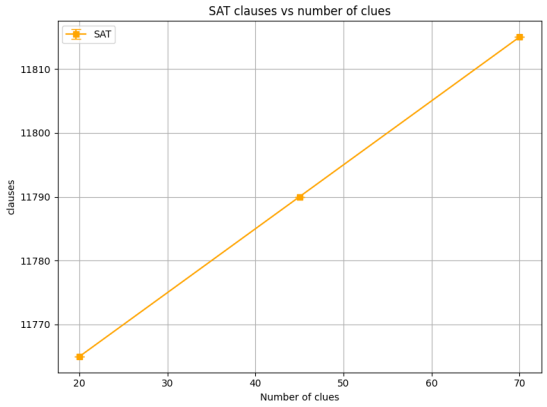

Figure 5.5: Number of SAT clauses as a function of the number of clues. Each dot represents the average time across puzzles. The number of clauses depends on the number of clues, this is because we always have for each rule from 1 to 6: $81+2916+2916+2916+number of clues+2916$. Indeed we have the number spanning from 11765 to 11815.

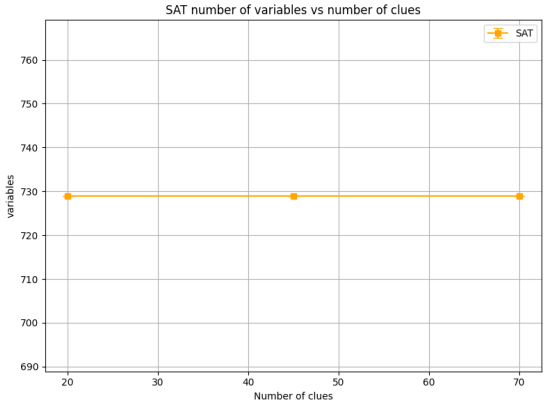

Figure 5.6: Number of SAT variables as a function of the number of clues. The variable count is constant because each possible digit in each cell is represented by a distinct propositional variable, independent of which cells are pre-filled. Only the number of clauses changes with the number of clues.

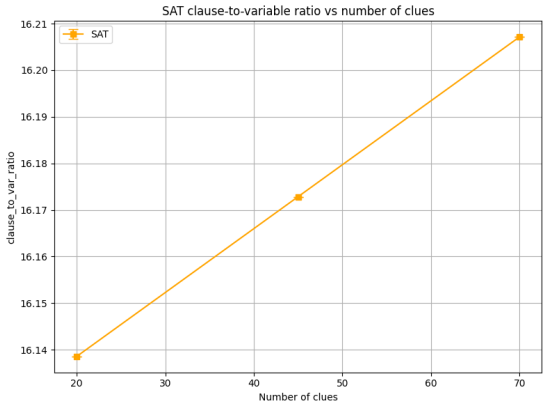

Figure 5.7: Clause-to-variable ratio in the SAT encoding as a function of the number of clues. The ratio increases with more clues because each pre-filled cell adds additional unit clauses to enforce its value, while the total number of variables remains constant. This reflects the growing density of constraints in the encoding as puzzles become more constrained.

6 How to Run

The code requires the following Python packages:

- **PySAT**: Provides the Glucose3 SAT solver (`pysat.solvers`). Install via pip:

  ```
  pip install python-sat[pblib,aiger]
  ```

- **NumPy**: For numerical operations (`numpy`). Install via:

  ```
  pip install numpy
  ```

- **Matplotlib**: For plotting metrics (`matplotlib.pyplot`). Install via:

  ```
  pip install matplotlib
  ```

- **os** and **time**: Standard Python library modules, included with Python.

Additionally, ensure that the helper modules `sudoku.py`, `a_star.py`, and `sat_solver.py` are located in the same folder as `tester.py` or accessible via the Python path.
Once dependencies are installed, you can run the ***tester.py*** file for replicating the experiments.
This will execute the tests, generate plots, and save csv results in the `my_results` folder.
Or for a **quick test** you can run the file ***main.py*** that will generate a random sudoku and solve it with A* and SAT. The output will look like this:

```
Generated Sudoku Puzzle:
. 1 . | . . . | . . .
. . . | . . 8 | 5 6 .
9 . . | 3 . . | . . 1
- - - + - - - + - - -
. . . | 6 . . | . . .
. . . | . . . | . . .
. 9 5 | . . . | . . .
- - - + - - - + - - -
. . 9 | 8 . 6 | . . .
. 7 . | 4 . 3 | . . .
. 6 8 | 7 . . | 4 . .

Sudoku solved with SAT:
8 1 6 | 5 7 9 | 3 2 4
2 3 7 | 1 4 8 | 5 6 9
9 5 4 | 3 6 2 | 8 7 1
- - - + - - - + - - -
1 8 2 | 6 3 4 | 9 5 7
7 4 3 | 9 5 1 | 2 8 6
6 9 5 | 2 8 7 | 1 4 3
- - - + - - - + - - -
4 2 9 | 8 1 6 | 7 3 5
5 7 1 | 4 2 3 | 6 9 8
3 6 8 | 7 9 5 | 4 1 2

Sudoku solved with A* in 61 steps.
8 1 7 | 9 6 5 | 3 4 2
4 3 2 | 1 7 8 | 5 6 9
9 5 6 | 3 4 2 | 8 7 1
- - - + - - - + - - -
7 8 4 | 6 9 1 | 2 5 3
6 2 3 | 5 8 7 | 9 1 4
1 9 5 | 2 3 4 | 7 8 6
- - - + - - - + - - -
5 4 9 | 8 2 6 | 1 3 7
2 7 1 | 4 5 3 | 6 9 8
3 6 8 | 7 1 9 | 4 2 5
```