

Artificial Intelligence – Homework Report

Name: Francesca

Surname: Cinelli

Student ID: 2046606

Introduction

This homework assignment focuses on the implementation of an entire pipeline for solving the selected game of Sudoku. Sudoku is an ideal candidate for this study because it represents a classic Constraint Satisfaction Problem (CSP) that is simple to define yet computationally challenging to solve as the puzzle difficulty increases.

To address this problem, I implemented two distinct Artificial Intelligence techniques. The first approach utilizes the A* (A-Star) search algorithm, a best-first search method that relies on heuristics to guide the exploration of the state space. The second approach involves a reduction of the Sudoku puzzle to a Boolean Satisfiability Problem (SAT), which is then solved using a modern SAT solver (Glucose3 via the PySAT library).

The solution was structured into two primary Python modules. The core logic, including the problem definition and the algorithms, are in a solver script, while the experimental execution and visualization logic are separated into a metrics script. This report details the high-level design of the implementation, the specific data structures employed, and a deep analysis of the experimental results. A significant part of this report will focus on two main questions I posed myself and answered: the behavior of the branching factor in A* and the distinct failure modes observed when the algorithms are presented with unsolvable instances.

Task 1: Problem

The first task required the modeling of the Sudoku puzzle within a generalized search framework. To comply with the requirement that the code must be generalized to all problems, I defined an abstract base class named *SearchProblem*. This class serves as an interface, mandating that any specific problem implementation must provide methods to retrieve the initial state, check for a goal state, generate successor states, and calculate a heuristic value. This design ensures that the underlying search algorithms remain agnostic to the specific domain they are solving; they interact only with the abstract methods of the *SearchProblem* class.

For the specific implementation of Sudoku, I created a class named *SudokuProblem* that inherits from *SearchProblem*. The state representation is a critical aspect of this implementation. A Sudoku grid is represented by a class called *SudokuState*. Internally, the grid is stored not as a mutable list, but as a tuple of integers. This design choice was necessary because the A* algorithm requires the use of a "closed set" (or explored set) to track visited states and prevent cycles. In Python, sets and dictionary keys must be hashable, and mutable lists do not satisfy this property. By using a tuple, the state becomes immutable and hashable, allowing for efficient $O(1)$ lookups in the explored set.

The initialization process involves parsing a string representation of the puzzle, where numbers represent pre-filled cells and dots or zeros represent empty spaces. The *is_goal* method is implemented straightforwardly by checking if the grid contains any zeros; if no zeros are present, the grid is complete. While a complete grid must also satisfy the row, column, and box constraints to be a valid solution, the validity is enforced incrementally during the successor generation phase rather than at the goal check. This ensures that the search tree never branches into an invalid state, significantly pruning the search space.

Task 2.1: Implementation of A*

The A* search algorithm was implemented to find the optimal solution path from the initial empty grid to the completed puzzle. The implementation strictly follows the standard A* pseudocode with duplicate elimination and no reopening of closed nodes.

The core data structure driving the algorithm is a priority queue, implemented using Python's *heapq* module. The frontier stores *Node* objects, which encapsulate the current state, the parent node, the action taken to reach this state, the path cost $g(n)$, and the heuristic value $h(n)$. The priority in the queue is determined by the f-score, where $f(n) = g(n) + h(n)$. This ensures that the algorithm always expands the node that is estimated to be closest to the goal.

To handle duplicate elimination, I utilized a set named *explored*. When a node is popped from the frontier, the algorithm immediately checks if its state is already in the *explored* set. If it is, the node is discarded. This expansion strategy is computationally efficient and prevents the algorithm from processing the same grid configuration multiple times, which would otherwise lead to infinite loops or redundant computations.

A critical component of the A* implementation for Sudoku is the *get_successors* function. A naive implementation might generate successors by finding the first empty cell and trying all numbers from 1 to 9. However, this results in a massive branching factor. Instead, I implemented the Minimum Remaining Values (MRV) heuristic within the successor generation logic. The algorithm scans the entire grid to identify the empty cell with the fewest legal moves. For example,

if one cell can only accept the number '5' while another cell can accept '1, 2, or 3', the algorithm chooses to fill the first cell. This drastically reduces the effective branching factor of the search tree.

For the heuristic function $h(n)$, I used the count of empty cells remaining in the grid. This heuristic is admissible because filling a cell constitutes one step with a cost of 1. Therefore, to solve a puzzle with k empty cells, the algorithm must make exactly k steps. Since the heuristic value exactly equals the remaining cost, $h(n)$ is essentially perfect for valid paths, although it does not account for the backtracking required if a dead end is reached.

Task 2.2: Implementation of Reduction to SAT

The second AI technique implemented was the reduction of Sudoku to the Boolean Satisfiability Problem (SAT). This approach translates the constraints of the puzzle into a propositional logic formula in Conjunctive Normal Form (CNF), which is then fed into the Glucose3 SAT solver provided by the *python-sat* library.

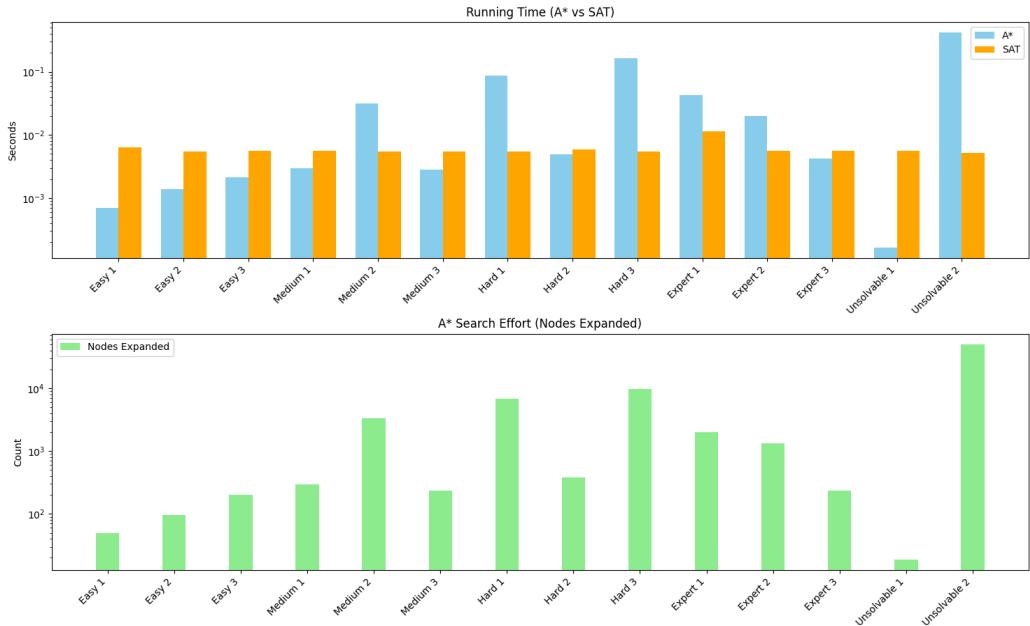
The modeling involves defining boolean variables and clauses. I mapped every possible decision in Sudoku to a unique integer variable. A variable $X_{r,c,v}$ is true if and only if the cell at row r and column c contains the value v . The mathematical mapping used to generate unique integer IDs for these variables is $ID = rN^2 + cN + v$, where N is the grid size (9). This bijection allows the solver to work with integers and allows the program to decode the integers back into a Sudoku grid upon finding a model.

The constraints were encoded into four types of CNF clauses. First, "Cell Definedness" clauses ensure that every cell must contain at least one number from 1 to 9. Second, "Cell Uniqueness" clauses ensure that no cell contains more than one number (mutual exclusion). Third, "Group Uniqueness" clauses enforce the Sudoku rules: for every row, column, and 3×3 box, and for every value v , there must be a cell containing v , and no two cells in the same group can contain v . Finally, "Pre-filled" clauses are added as unit clauses (clauses with a single literal) for the initial numbers given in the puzzle, forcing the solver to respect the initial state.

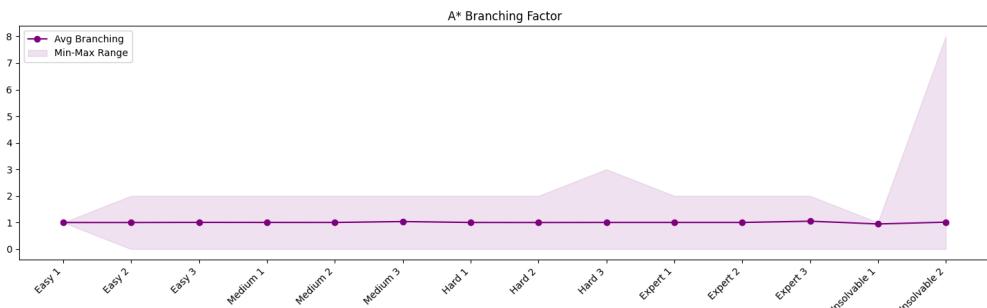
The solver is integrated directly into the *SudokuSAT* class. The method *solve()* generates the clauses, feeds them to the Glucose3 solver, and checks for satisfiability. If the instance is satisfiable, the solver returns a model (a list of truth assignments), which is then decoded back into the 9×9 grid format. This approach relies on unit propagation and clause learning to find solutions.

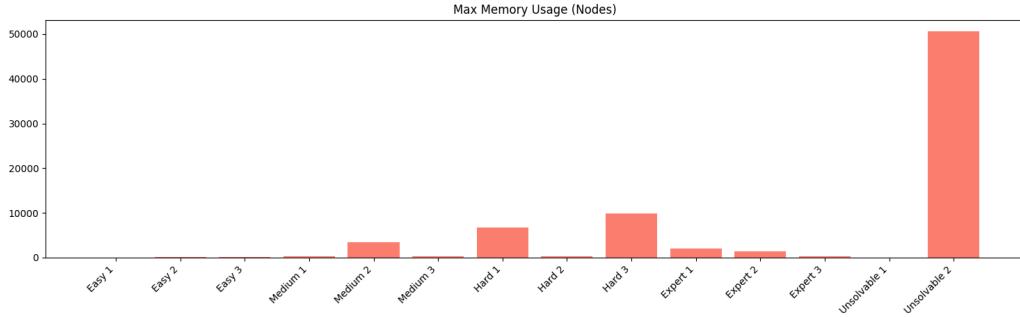
Task 3: Experimental Results

To validate the efficiency of the implemented algorithms, I conducted a series of benchmarks on a dataset comprising 15 Sudoku puzzles of varying difficulty, ranging from "Easy" to "Expert" (using Norvig's and Inkala's famous hard instances), and including two specifically designed "Unsolvable" corner cases.



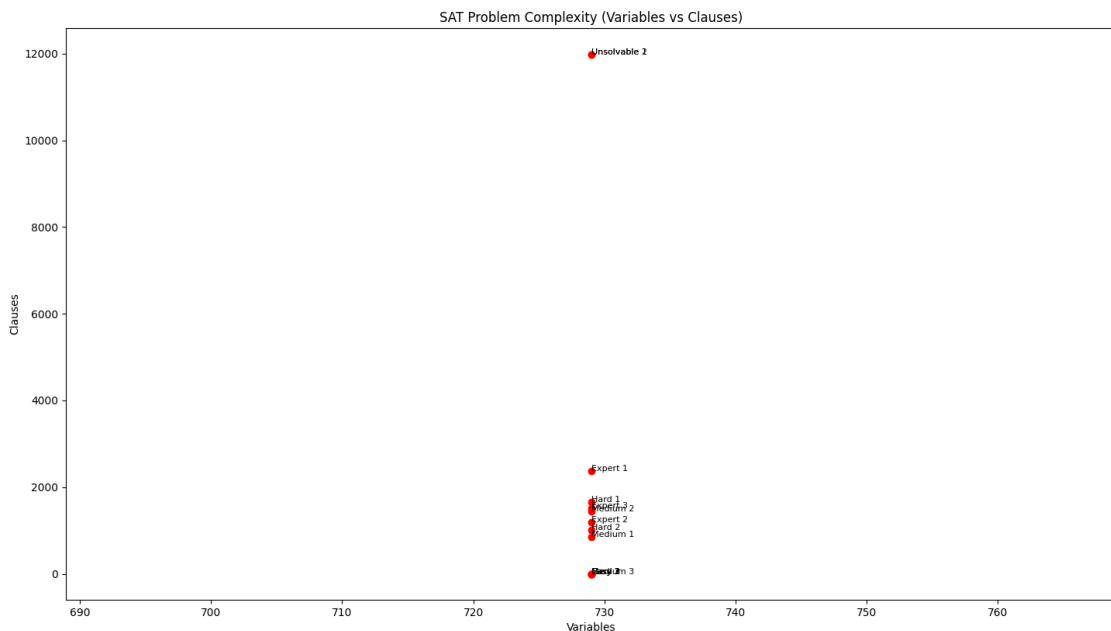
The first metric analyzed was the running time. As illustrated in the running time plot, the SAT solver (orange bars) exhibits relatively constant and extremely fast performance across almost all difficulty levels, typically solving instances in fractions of a second. In contrast, the A* algorithm (blue bars) shows high variance. For easy and medium puzzles, A* is competitive, but for the "Unsolvable 2" instance (an empty grid with a conflict), the time explodes. This highlights a fundamental difference: SAT solvers use conflict-driven clause learning to prune vast parts of the search space effectively, while A* struggles when the heuristic does not provide strong guidance. The plot also visualizes the number of expanded nodes, which correlates perfectly with the time taken; hard puzzles require expanding thousands of nodes, while the SAT solver's "decisions" (shown in later plots) remain manageable.

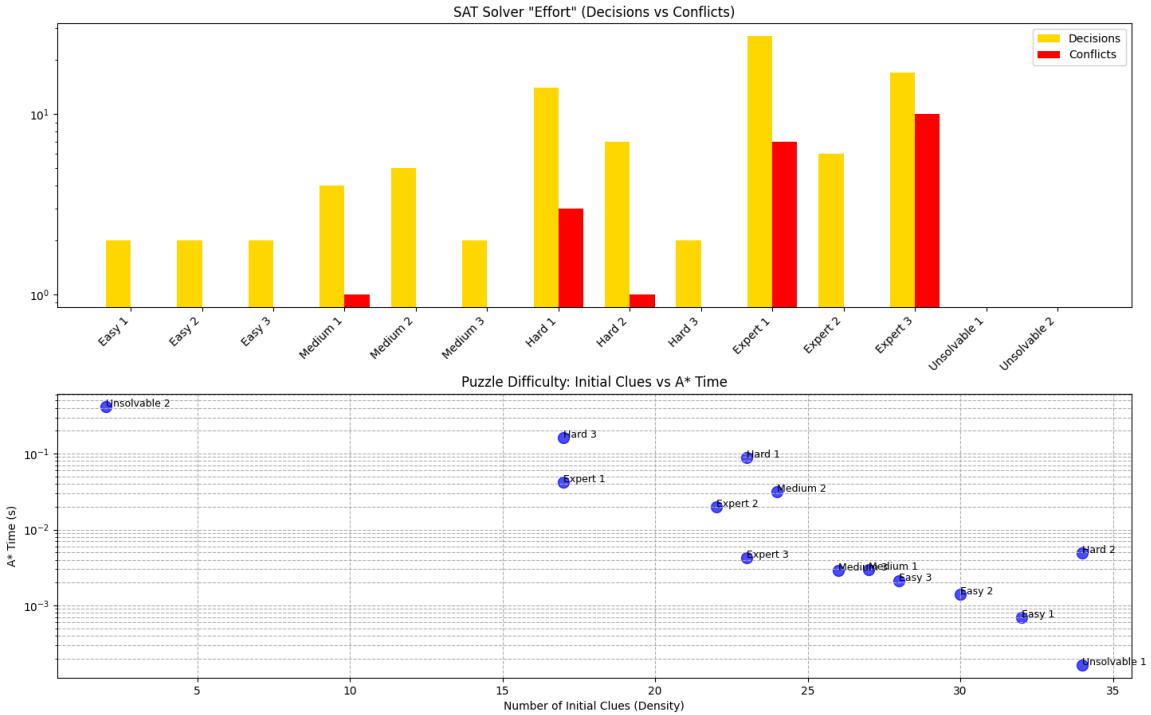




A particularly interesting metric captured is the Branching Factor of the A* search. Looking at the "Average Branching" curve, one might be surprised to see that it hovers extremely close to 1.0 for almost all solvable puzzles.

Why is the Average Branching Factor approx. 1? This is the first main question I posed myself for this report. This phenomenon is a direct result of the Minimum Remaining Values (MRV) heuristic implemented in the successor function. Sudoku is a highly constrained problem. In the vast majority of steps, especially after a number is placed, logical propagation implies that one or more subsequent cells have only one legal possibility. When the algorithm identifies a cell with only one legal move, it executes a "forced move." In this case, the node has only 1 successor. Even in difficult puzzles where the algorithm must guess (branching into 2 or 3 paths), that single guess often triggers a cascade of 10 or 20 forced moves. Mathematically, if we have one branching step of size 2 followed by 19 steps of size 1, the average is $\frac{2+19}{20} = 1.05$. In the visualization, this creates a line that appears nearly flat at 1. This confirms that the A* implementation is behaving intelligently, deducing moves rather than blindly guessing.





The analysis of the "Unsolvable" corner cases provided the most critical insight into the algorithmic behaviors. This is where I posed my second question: Why is there a difference in solving time for the A* algorithm for two instances that are both unsolvable and that both present a conflict in the first row?

The "Unsolvable 1" instance represents a dense grid (many clues) that contains an immediate logical contradiction, such as two 5s in the first row. When A* starts, it attempts to generate successors for the initial state. The MRV heuristic scans the board and calculates the valid moves for empty cells. Because of the immediate conflict and the constraints of the pre-filled numbers, the constraint propagation logic determines that there is at least one empty cell with zero legal values. Consequently, the `get_successors` function returns an empty list. The frontier becomes empty immediately, and the algorithm correctly terminates with an "Unsolvable" status in near-zero time.

The "Unsolvable 2" instance is a very sparse grid (mostly empty) with a subtle conflict in the first row. Here, A* fails catastrophically. The heuristic function $h(n)$ counts empty cells. As the search proceeds, the cost $g(n)$ increases by 1 for every number placed, and $h(n)$ decreases by 1. This results in a constant f-score ($f = g + h$) for every node in the search tree. When all nodes have the same priority, the Min-Heap behaves like a Queue, turning the smart A* search into a Breadth-First Search (BFS). Since the board is empty, the constraints are loose, and the branching factor is high (initially 9 choices per cell). The algorithm attempts to fill the grid layer by layer, generating an exponential number of nodes (9,81,729...) before it ever reaches the depth where the conflict becomes apparent. The search space explodes, filling the memory and hitting the node limit, resulting in a "Timeout" status instead of an "Unsolvable" status.

How to Run

To reproduce the experimental results presented in this report and verify the algorithmic implementation, a standard Python 3 development environment is required. The project has been designed for simplicity and portability, minimizing external dependencies while ensuring robust functionality for both the search algorithms and the visualization tools. In fact, it was initially written as a Google Colab Notebook to verify the main logic and then transferred to a standard python environment in VS Code where minor modifications were made and the metrics logic was implemented.

Dependencies and Environment Setup

The primary prerequisite for executing the code is a working installation of Python 3. The solution relies on the *python-sat* library (specifically the *pysat* module) to interface with the Glucose3 SAT solver, and the *matplotlib* library to generate the performance graphs discussed in the section “Task 3: Experimental Results” of this report. While the core A* logic utilizes only standard Python libraries (*heapq*, *time*, *math*), the SAT reduction cannot function without *python-sat*.

To replicate the exact environment used for these experiments, you may install the necessary packages individually via the terminal commands *pip install python-sat* and *pip install matplotlib*. Alternatively, given that a *requirements.txt* file is provided with the submission, the entire dependency tree can be installed in a single step using *pip install -r requirements.txt*.

Execution Instructions

The codebase is organized into two distinct executable scripts, each serving a specific purpose in the evaluation pipeline.

1. Running the Solvers (Single Instance): The script named *sudoku_solver.py* contains the complete implementation of the generalized *SearchProblem* interface, the A* algorithm, and the SAT reduction logic. This file is intended for verifying the correctness of the algorithms on individual puzzles.

To run this script, execute the command *python3 sudoku_solver.py* in your terminal. By default, the *main* block of this script is configured to solve a classic "Hard" benchmark instance from Peter Norvig's dataset. It will output the initial grid, the solved grid, and key performance metrics such as execution time and nodes expanded. To test the solver against different puzzle instances, you can modify the *hard_puzzle* string variable (following the section “Formatting Input Data” of this report) within the *if __name__ == "__main__":* block of the code and re-run the command.

2. Reproducing Experimental Metrics: The second script, *sudoku_metrics.py*, is designed to reproduce the full experimental campaign described in the "Task 3: Experimental Results" section of this report. This script imports the solver logic from the main module and executes a batch benchmark across 15 distinct Sudoku puzzles, ranging from "Easy" to "Unsolvable."

To generate the statistics and visualizations, execute the command `python3 sudoku_metrics.py`. This process will sequentially solve all configured instances using both A* and the SAT solver. Upon completion, it will output a textual summary of the results to the console (including an example of a solved instance between the ones given in input) and automatically generate the comparative plots (Running Time, Branching Factor, and SAT Complexity) using *matplotlib*. If you wish to expand the benchmark suite or test specific edge cases, you can modify the *puzzles* list inside the *run_benchmarks* function (following the instructions in the following section of the report).

Formatting Input Data

For manual testing using *sudoku_solver.py* and *sudoku_metrics.py*, inputs must be provided as a flattened string of 81 characters. The format follows row-major order (reading the grid from top-left to bottom-right). Use digits '1'-‘9’ for known numbers and either ‘0’ or ‘.’ for empty cells.

For example, consider the following grid state:

			2	6		7		1
6	8			7			9	
1	9				4	5		
8	2		1				4	
		4	6		2	9		
5					3		2	8
		9	3				7	4
	4			5			3	6
7		3		1	8			

The corresponding input string would be:

“...26.7.168..7..9.19...45..82.1...4...46.29...5...3.28..93...74.4..5..367.3.18...” or
“000260701680070090190004500820100040004602900050003028009300074040050036703018000”

This standardized format ensures that any Sudoku configuration, regardless of its visual representation, can be easily processed by both the A* and SAT solver implementations.

Code Improvements

Although the code works perfectly for all imaginable instances, there are some improvements or details that need to be mentioned.

Generalization to Variable Grid Sizes

One of the core strengths of the *SearchProblem* interface design is its ability to generalize beyond the standard 9×9 Sudoku format. The current implementation was constructed with a dynamic size parameter N , allowing the solver to theoretically handle grids of any dimension, such as small 4×4 puzzles or large 16×16 "Hexadoku" instances. However, scaling the grid size introduces specific constraints and required modifications to the logic, particularly regarding the definition of sub-grid (box) constraints. The following are all the possible cases:

1. Perfect square grids (e.g. 4×4 , 16×16 , 25×25): for grid sizes where N is a perfect square (i.e. \sqrt{N} is an integer), the current code is fully compatible without logical modification. For example, to solve a 16×16 puzzle, one would simply instantiate the problem with the explicit size parameter:

`SudokuProblem(puzzle_string, n=16)`. The input string length would need to be $N^2 = 256$ characters. Given that the current implementation derives the box size via `int(math.sqrt(n))`, for $N = 16$, this correctly identifies a 4×4 sub-grid. The constraint logic in both A* (`_get_legal_values`) and SAT (`generate_clauses`) would automatically enforce uniqueness within these 4×4 blocks.

The downside is that moving to 16×16 significantly increases the state space. The branching factor potentially grows from 9 to 16, and the search depth increases from 81 to 256. While the SAT solver handles this scaling robustly due to its efficient clause learning, the A* algorithm would likely require more advanced heuristics (e.g. "Most Constrained Variable" tie-breaking) to avoid timeouts on hard instances. Conversely, 4×4 grids are trivial and would be solved in negligible time (<0.001s).

2. Non-square grids (e.g. 6×6 , 10×10): standard Sudoku variants often exist on grids where N is not a perfect square. For example, a 6×6 grid typically uses rectangular sub-grids of size 2×3 . The current implementation assumes square sub-grids ($k \times k$). If $n=6$, the code calculates `box_size = int(sqrt(6)) = 2`. It effectively attempts to enforce constraints on 2×2 boxes, which is geometrically invalid for a width of 6 (as 2 does not divide 6 into equal square regions, nor does it cover the 2×3 standard layout). To support these geometries, the code would need to be refactored to accept separate parameters for `box_height` and `box_width` (e.g. $h=2, w=3$ for $N=6$). The SAT clause generation and A* legal value checking would then need to iterate over r in steps of `box_height` and c in steps of `box_width`.

3. Prime dimension grids (e.g. 5×5 , 7×7): for grid sizes that are prime numbers, standard "box" constraints are mathematically impossible (you cannot divide a 5-width row into equal sub-grids). These puzzles are usually treated as Latin Squares, where only row and column constraints apply. Supporting these would require a "flag" in the constructor (e.g. `use_boxes=False`). If enabled, the logic would simply skip the "Box Uniqueness" loops in the SAT generation and the box checking in the A* successor function. This would actually simplify the problem constraints, although it might make the search tree "looser" (higher branching factor) due to fewer constraints pruning the possibilities.