SAT Solvers

Aanya Garg

October 7, 2025

1 Introduction

People have always dealt with problems. As time went on, those problems became less about survival and more about ideas: how to send information across the world, how to create systems that shape the way we live, and how far the power of technology can really go. However, when addressing these problems, we always face the same two fundamental questions: can we solve this and if so, how fast can we do it?

That brings us to one of the most famous problems in mathematics and computer science: the Boolean Satisfiability Problem (SAT). It asks whether a formula composed of Boolean variables can be satisfied by some assignment of TRUE/FALSE values. This problem is not just another abstract puzzle, as it was the first proven NP-complete problem. This means SAT is a gateway problem: if an efficient algorithm is found for SAT, it would immediately provide efficient algorithms for thousands of other problems in scheduling, optimization, verification, and beyond.

Currently, we know how to solve SAT in principle through brute force, as well as some sophisticated algorithms for special cases, but do not have an efficient algorithm for all instances. In this project, we will study SAT by coding several SAT Solvers. Some algorithms are classics that researchers have been studying for decades, and others will be ones we try ourselves. We will then test these algorithms on known sets of complex logical formulas to see how they perform. Our goal is to see which ones actually work best, both in terms of always finding the right answer and in how quickly they do it.

2 Background

2.1 What is SAT?

The Boolean Satisfiability Problem (SAT) is one of the most famous problems in mathematics and computer science. In SAT, there is a logical formula made up of variables connected by

AND ($\wedge$), OR ($\vee$), and NOT ($\neg$). The question is whether one can assign TRUE or FALSE values to the variables such that the whole formula comes out to TRUE.

For example, consider the following:

$$(x \vee \neg y \vee z) \wedge (\neg x \vee y)$$

Is it possible to find a combination of TRUE and FALSE values for x, y, and z so the formula is satisfied? In this case, the answer is yes, it is satisfiable with multiple possible solutions. One way is setting x to TRUE, y to TRUE, and z to FALSE. Another is setting x to FALSE, y to TRUE, and z to TRUE. However, this is not true for all formulas. Take, for example:

$$(x) \wedge (\neg x)$$

No matter what value is assigned to x, one of the clauses is guaranteed to be FALSE so the whole formula can never be TRUE. Thus, it is unsatisfiable. This demonstrates the variability of the SAT problem, some formulas work with tons of solutions, while others are simply impossible.

2.2 Complexity-Theoretic Significance

At first glance, this may seem like nothing more than a brain teaser or puzzle. However, SAT is much more than that. In 1971, Stephen Cook showed that SAT was the first NP-complete problem, or that SAT is as difficult as thousands of other problems across many different fields. That result, known as Cook's Theorem, has two important results:

1. If there were an efficient algorithm to solve SAT, then all problems in NP could be solved efficiently.
2. The question of whether such an algorithm exists is equivalent to the famous open problem of whether $P = NP$.

Certain cases of SAT have different complexity. Take 2-SAT for example, where each clause has at most two literals, and be solved in polynomial time using graph algorithms. On the other hand, 3-SAT remains NP-complete and serves as the traditional "hard" version of the problem.

2.3 Historical Development of Solvers

Currently, we know how to solve SAT using the following methods:

- Brute force: The most straightforward approach is to try possible assignment of TRUE/FALSE values. While guaranteed to work, it is extremely slow as the number of assignments grows exponentially with the number of variables.
- DPLL (Davis–Putnam–Logemann–Loveland): A recursive method that improves efficiency by focusing on eliminating impossible cases using strategies like unit propagation and pure literal elimination.
- Modern SAT solvers: Extensions of DPLL, such as CDCL (Conflict-Driven Clause Learning), add techniques like clause learning, backjumping, and randomized restarts. These allow SAT solvers to handle large instances with millions of variables, although it is still theoretically NP-complete.

2.4 Applications of SAT

The importance of SAT extends far beyond just theory. The implication of being NP complete means is that if an efficient algorithm for SAT existed, it would instantly give efficient algorithms for a wide range of real-world problems, including:

- Scheduling airplane flights
- Designing and verifying computer chips
- Solving puzzles like Sudoku
- Tackling optimization problems such as the Traveling Salesperson Problem

Even without a general efficient algorithm, however, SAT solvers are already widely used in hardware and software verification, artificial intelligence, cryptography, and operations research. Although the worst case scenario SAT is NP-complete, modern SAT solvers handle industrial-scale problems with millions of clauses with the help of heuristics, preprocessing, and clever algorithmic engineering.

3 Methods

3.1 Algorithms Implemented

In order to study the performance of SAT solvers, I implemented a range of algorithms that vary from naive to advanced.

DPLL (Davis–Putnam–Logemann–Loveland) Algorithm This classical recursive algorithm uses backtracking, unit propagation, and pure literal elimination to prune large parts of the search space. I implemented DPLL in Python to serve as a "baseline" structured solver.

Heuristic Variable-Picking Algorithms I tested custom heuristics that decide the order in which variables are assigned in "smart" ways

1. Choosing the variable that appears most frequently. One of the simplest but still effective deterministic heuristics is to select the variable that appears most frequently across all clauses. Assigning them early can immediately simplify a large portion of the instance, either by satisfying many clauses or by reducing their size.

2. Clause-Weighted Variable Selection. This strategy adds importance to variables based on the length of clauses they appear in. The main idea is that shorter clauses are more restrictive, since they have fewer opportunities to be satisfied. To capture this, each clause is given a weight equal to the inverse of its length. Then, for each variable, we sum the weights of all clauses in which it appears. The solver then selects the variable with the highest weighted score.

3. Adaptive Probabilistic Hybrid. This strategy blends deterministic selection with randomness depending on the search depth. At the top of the search tree, where choices strongly influence the rest of the solution, the solver favors deterministic rules (such as those in the first two "smart" ways: frequency and constraints). Deeper in the tree, however, it introduces more randomness as those rigid strategies could lead to dead end scenarios.

3.2 Implementation Details

All solvers were implemented in Python 3 using only standard libraries (with Pandas and Numpy used for data collection and analysis).

- Input Format: SAT instances were generated in Conjunctive Normal Form (CNF). Each formula was represented as a list of clauses, where each clause was a list of integers corresponding to literals (e.g., x represented by 1, $\neg x$ by -1).

- Correctness Checks: Each solver was tested against small formulas with known solutions to confirm being correct before performance testing.

3.3 Testing Framework

To compare solvers, I used the following testing environment:

- Instance Types: Random 3-SAT (uf20, uf50) and structured SAT (pigeonhole, graph coloring, Towers of Hanoi).
- Metrics: Runtime (seconds, with 30s timeout), Correctness, Recursion failures.
- Hardware & Software: MacBook Air (Apple M3, 2024) with 8-core CPU and 8 GB RAM.

4 Results

4.1 Overview

The experiments revealed distinctions between solver performance on random versus structured benchmarks. While all solvers handled small random instances with ease, larger or more structured formulas exposed significant weaknesses, with both exponentially large slowdowns and practical implementation limits such as recursion depth. Two forms of failure were reported: timeouts, where a solver exceeded the 30 second limit, and recursion limit exceeded, where the solver reached Python's maximum recursion depth despite extending it to 20,000.

4.2 Random 3-SAT Formulas

The experiments on random 3-SAT formulas (uf20 and uf50 benchmarks) show a divide between the smaller uf20 instances and the larger uf50 ones.

For uf20 problems, all solvers were able to find solutions quickly, usually in under 10 milliseconds. DPLL consistently produced results in the 1–8 ms range, while the frequency and weight-based heuristics sometimes added extra time but remained efficient. The hybrid solver's runtimes were also competitive, generally close to the deterministic heuristics. These results suggest that on small random formulas, even simple DPLL is sufficient, and heuristic improvements do not provide dramatic speedups.

For uf50 problems, the differences between the solvers became more apparent. On many uf50 instances, DPLL still returned results in 1–10 seconds, but the frequency heuristic sometimes required longer runtimes (up to 12 seconds), reflecting the time lost by evaluating variable counts in larger formulas. The weighted heuristic, however, was much more efficient, often solving those same instances in under 1 second. The hybrid strategy was inconsistent: on some formulas it matched or outperformed DPLL, but on others it ran much longer or even timed out at the 30 second limit.

Overall, the data shows that weighted clause heuristics scale better than simple frequency counting when formula size increases. Additionally, while DPLL remains competitive on

smaller instances, it loses efficiency as variable counts and clause density grow. Hybrid methods show potential but also instability.

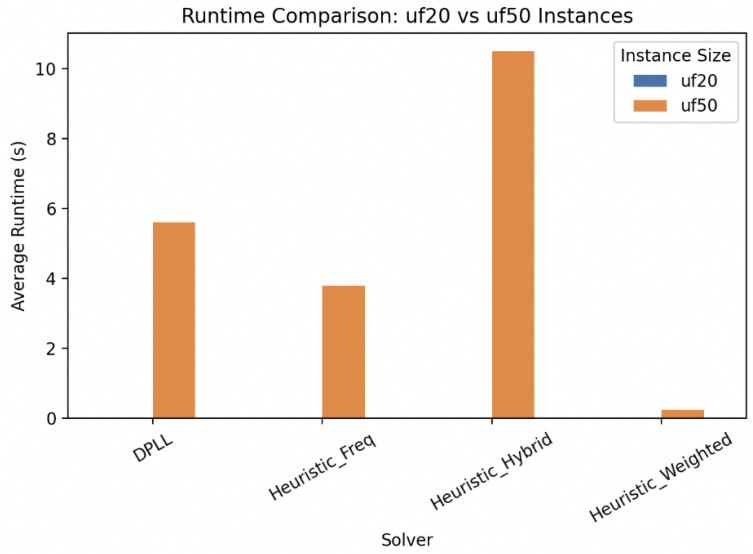

Figure 1: Average runtime comparison between uf20 and uf50 random 3-SAT instances.

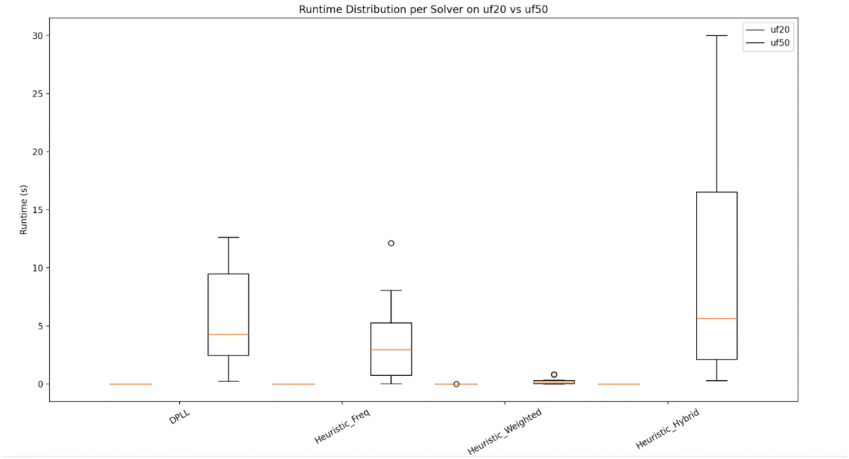

Figure 2: Runtime distribution of solvers on uf20 vs uf50 random 3-SAT instances.

4.3 Hard Problems

The structured formulas produced very different results compared to the random ones. For the pigeonhole problems, all larger ones (hole8, hole9, hole10) hit the 30 second timeout for every solver. The smaller ones had more variation: hole6 was solved quickly by every solver, and hole7 was solved within 15 seconds by the frequency and weighted heuristics, while

DPLL and hybrid timed out. Since pigeonhole encodings are unsatisfiable, these results show how much harder it is for the solvers to prove unsatisfiability versus just finding a satisfying assignment.

The Hanoi tower encodings were also very hard. Even for hanoi4 and hanoi5, every solver timed out before finishing. The deep recursive nature of these formulas makes them difficult, and the branching heuristics did not help.

Finally, the graph coloring instances (g125, g250) were the hardest overall. All solvers reached the timeout without producing a result. These formulas caused very deep recursion, and I had to add a recursion limit on top of the timeout wrapper to keep the program from crashing.

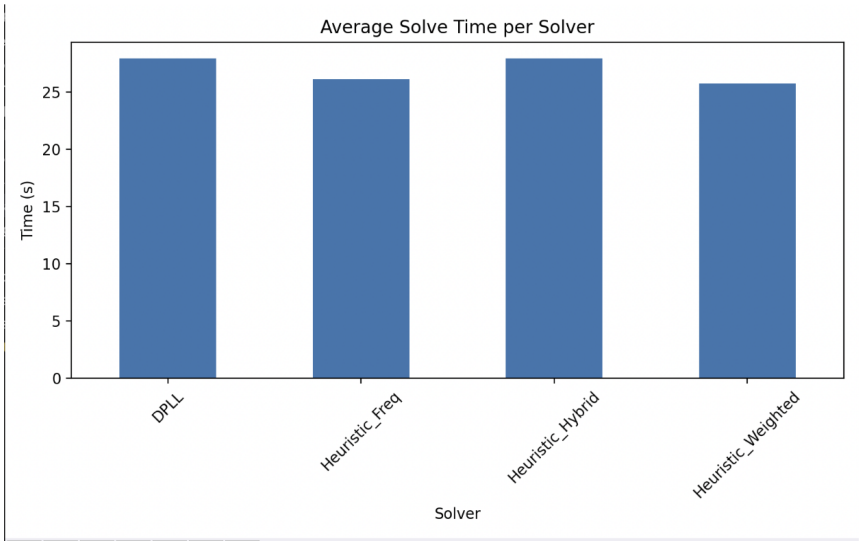

Figure 3: Average runtime of each solver across hard benchmark instances (pigeonhole, Hanoi, and graph coloring).

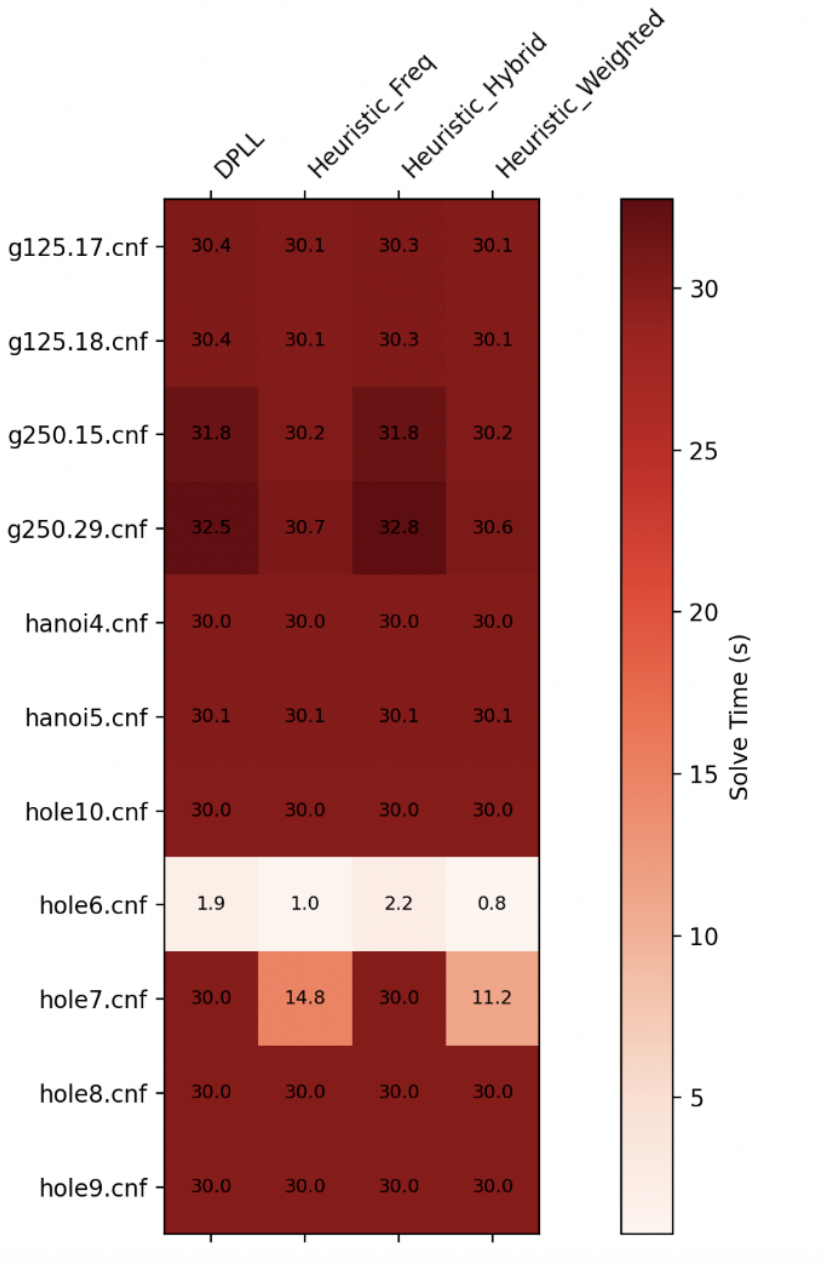

Figure 4: Heatmap of solver runtimes across structured SAT problems.

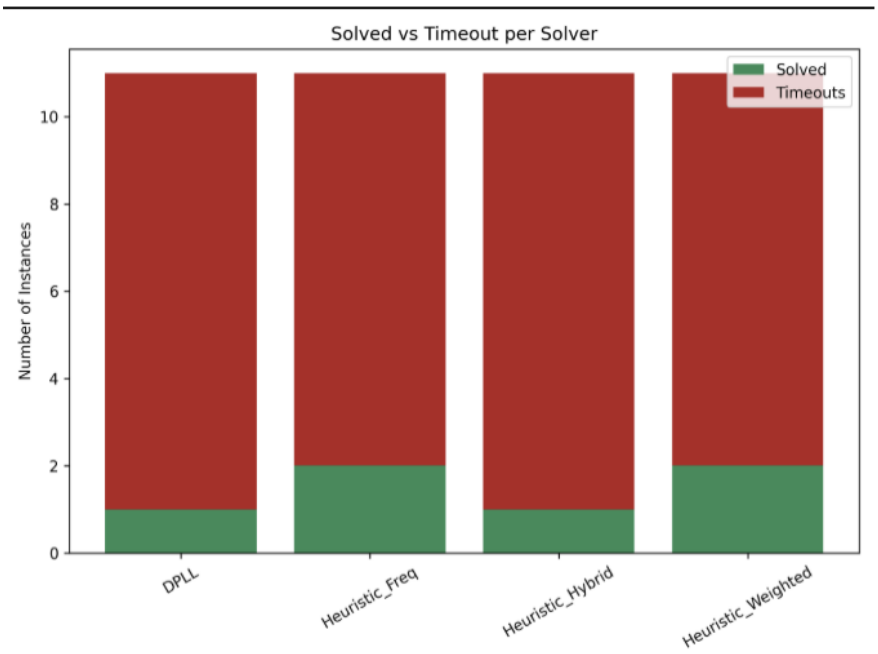

Figure 5: Solved vs Timeout per solver across structured benchmarks.

4.4 Timeout and Recursion Limit Behavior

A notable distinction emerged between timeouts and recursion limit failures.

- Timeouts indicate that the solver explored too broad a search space without effectively eliminating cases. These were common in uf50 instances with poor heuristics and in large hard problems.
- Recursion limit failures reflect search trees that are extremely deep but narrow. These were most common in Hanoi and large graph-coloring encodings.

4.5 Summary

Overall, the experiments confirm that:

1. Baseline DPLL remains surprisingly competitive on random benchmarks.
2. Frequency and weighted heuristics help in some cases but introduce variability that can worsen performance on structured instances.
3. Structured formulas expose the limits of recursive Python implementations, producing recursion depth overflows even when the system recursion limit is raised substantially.

5 Discussion

The results show the difference between random and structured SAT problems, as well as the limits of simple SAT solvers written in Python. For the random 3-SAT formulas, the solvers performed well on the smaller uf20 instances and sometimes on the uf50 ones, though the heuristic design did make a clear difference in runtime. These results suggest that even lightweight heuristics like frequency and weight can demonstrate meaningful improvements in some scenarios.

The structured formulas, on the other hand, had different results. For the pigeonhole encodings, solvers could only handle the very smallest cases before taking too much time. The Hanoi encodings had even worse results, as the deep recursion made them practically unsolvable within the limits of our implementation. Graph coloring problems also caused consistent timeouts, showing that these constraint structures were difficult for all the solvers. These outcomes demonstrate how the structure of the formula is just as important as the size. Random formulas tend to give solvers more "easy wins," while structured problems expose their weaknesses.

Another important observation was the need for both a timeout and recursion guard. Without these, the solvers would either run indefinitely or crash, which would make testing on a large scale very difficult on a personal computer. These controls also revealed where different families of problems fail: timeouts for breadth-heavy formulas, recursion errors for depth-heavy ones.

Overall, this project shows that while simple heuristics can improve performance on random instances, they are not enough to break through structured SAT problems. This gap suggests a direction for future work with more advanced heuristics, hybrid strategies, or even machine learning based strategies that could be explored. In practice, this is why modern SAT solvers use clause learning from conflicts, more sophisticated branching, and preprocessing to go beyond what was tested here.

References

Cook, S. A. (1971). The complexity of theorem-proving procedures. Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC '71), 151–158. https://doi.org/10.1145/800157.805047

Davis, M., Logemann, G., & Loveland, D. (1962). A machine program for theorem-proving. Communications of the ACM, 5(7), 394–397. https://doi.org/10.1145/368273.368557

Hoos, H. H., & Stützle, T. (2000). SATLIB: An online resource for research on SAT. In SAT 2000: Highlights of Satisfiability Research in the Year 2000 (pp. 283–292). IOS Press. http://www.cs.ubc.ca/~hoos/SATLIB/

Yurichev, D. (2025). SAT/SMT by Example. https://smt.st/SAT_SMT_by_example.pdf

Soos, M. (2018). Modern SAT solvers: fast, neat and underused (part 1 of N). https://codingnest.com/modern-sat-solvers-fast-neat-underused-part-1-of-n/