Analyzing SAT Solver Performance: From NP-Completeness to Predictive Runtime Models.

Author: Aanya Garg
Mentor: Dr William Gasarch

November 9, 2025

1 Introduction

Throughout history, humanity has dealt with problems. As time passed, those problems evolved from survival to modern ideas: how to send information across the world, how to create systems that shape our lives, and how far the limits of computation can be pushed. Yet, every challenge somehow still returns to two essential questions:

1) CAN we solve this problem and

2) If so, how FAST can we do it?

This leads 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 far more than another abstract puzzle. It was the first proven NP-complete problem. 'NP' stands for Nondeterministic Polynomial time: the class of problems that are difficult to solve, but whose solutions can be verified easily. For instance, the Sudoku puzzle may be difficult to solve, but once a solution is presented it can be checked almost instantly. Furthermore, all NP-complete problems can be transformed into one another. Thus, SAT is a gateway problem: if an efficient algorithm is found for SAT, it would immediately provide efficient algorithms for all other NP complete problems.

Take the famous traveling salesman problem. Imagine you are a traveling salesperson, visiting many cities and trying to find the shortest route that visits those cities and returns back to your starting point. This is a difficult task as the number of paths increases exponentially with the number of cities that are desired to visit. However, if there were an efficient algorithm to solve SAT, as both problems are NP-complete, you could convert this traveling salesman problem into an SAT problem and then solve that instead. In other words, a

breakthrough in SAT solving would unlock fast solutions to countless other real-world challenges, from scheduling flights and optimizing delivery routes to verifying computer circuits and designing efficient networks.

One may wonder why we cannot simply use a brute force approach executed by a supercomputer, but even the fastest computers cannot keep up with how quickly the problem grows. For every variable added to the equation, the number of possible truth assignments doubles. That means that if a supercomputer could check a trillion combinations per second, a formula with just 100 variables would have 2^{100} combinations possible. That would take more than 30 billion years to finish—longer than the age of the universe! So, no matter how powerful our hardware gets, brute force remains impractical once instances become large, which is why researchers focus on finding more efficient algorithms instead of faster machines.

In this project, we will study SAT by coding several algorithms to solve it. Some algorithms are classics that researchers have been studying for decades, used as benchmarks, and others will be ones that we come up with 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 accuracy and in how quickly they do it.

2 Background

2.1 How are SAT formulas structured?

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. In SAT, some formulas work with many solutions, while others are simply impossible.

2.2 Complexity-Theoretic Significance

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 consequences:

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 $P = NP$ (where P is the class of problems that can be solved efficiently in polynomial time, and NP is the class of problems whose solutions can be verified efficiently in polynomial time).

Certain cases of SAT have different complexity. Take 2-SAT, for example, where each clause has at most two literals and can 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 Current Applications of SAT

Even without a general efficient algorithm, 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 by translating them into logical formulas that the solver can process.

For instance, verifying that a computer chip functions correctly can be expressed as checking whether a logical formula representing the circuit is satisfiable. Although SAT is NP-complete in the worst case, the ability to represent various problems as logical constraints makes solvers extremely powerful across many industries.

3 Methods

3.1 Algorithms Implemented

In order to study the performance of SAT solvers, we implement a range of algorithms.

DPLL (Davis–Putnam–Logemann–Loveland) Framework This famous recursive framework uses backtracking, unit propagation, and pure literal elimination to prune large parts of

the search space. This was used as the structure to implement all solvers in this project. To serve as a "baseline" structured solver, we begin with the first unassigned variable. **If any algorithm is referred to as 'DPLL' in this project, it is the baseline DPLL being referenced** (all algorithms are technically using the DPLL framework, just with different "smart" ways to choose a first variable).

Heuristic Variable-Picking Algorithms Custom heuristics that decide the order in which variables are assigned in the following "smart" ways:

1. **Frequency-Based Selection:** Choosing the variable that appears most frequently first. Assigning it 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 idea is that shorter clauses are more restrictive, since they have fewer opportunities to be satisfied. To model 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 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, it introduces more randomness as rigid strategies could lead to dead end scenarios (where the solver moves into unsatisfiable branches but we do not know until after going through scenarios unnecessarily).

3.2 Implementation Details

All solvers were implemented in Python 3 using only standard libraries (with Pandas, Numpy, Scipy, and Matplotlib 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, we use the following testing environment:

- **Instance Types:**
  - **Random 3-SAT:** Instances such as uf20-xxxx.cnf and uf50-xxxx.cnf were taken from the SATLIB library. These varied in clause-to-variable ratios and are well known for producing both easy and hard cases depending on density.
  - **Structured SAT Instances:** Additional benchmarks were included to test solver performance on formulas that represented problems known to be "hard." These included:
    * **Pigeonhole principle (hole.cnf):** Encodes the unsatisfiable problem of placing n+1 pigeons in n holes. These are guaranteed unsatisfiable and require significant search.
    * **Graph coloring (g.cnf):** Encodes the k-colorability of graphs, often yielding satisfiable but nontrivial instances.
    * **Towers of Hanoi (hanoi.cnf):** Encodes the Towers of Hanoi problem into CNF, creating structured recursive search instances.
- **Metrics Measured:**
  - **Runtime (seconds):** Time to determine satisfiability.
    * **Timeout Handling:** To prevent the solvers from running indefinitely on harder instances, each solver call was wrapped in a five-minute timeout. If a solver exceeded this limit, the run was terminated and recorded as `TIMEOUT` with a runtime of 300.0 seconds.
  - **Correctness:** Whether the solver returned the right answer.
- **Hardware & Software Environment:** All experiments were run on a MacBook Air (Apple M3, 2024) with 8-core CPU and 8 GB RAM. Timing was measured using Python's built-in `time` library.
- **Data Collection & Visualization:** Results were stored in Pandas dataframes and visualized using Matplotlib to generate plots of runtime versus input size, solver comparisons, and clause density effects.

3.4 Data Analysis and Modeling

To analyze solver performance trends, we model runtimes from each solver as a function of problem size (measured by the number of variables). Since SAT problems are expected to exhibit exponential time complexity, we fit the runtimes $T(n)$ to the model $T(n) = ae^{bn}$, where a is a scaling constant and b represents the exponential growth rate. This was done separately for each solver using Python's SciPy `curve_fit` function.

To better visualize exponential growth, all runtime data were plotted on a logarithmic scale. Applying the logarithm transforms the exponential relationship into a linear one for ease of observance. In this form, the slope of the line corresponds to the exponent b, allowing us to compare solvers based on how quickly their runtimes increase as problems grow in size. Smaller b values indicate more efficient scaling behavior.

This analysis was done to achieve a quantitative view of heuristics used, linking observed runtime patterns to the theoretical expectations of NP-complete problems.

4 Results

4.1 Overview

The experiments revealed distinctions between the performance of the solver on random versus structured benchmarks. Although 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. The results were accurate across all benchmarks: instances were either solved successfully within the cap or recorded as timeouts. In a few cases, runtimes extended well past the 300-second cut-off into the thousands of seconds, showing just how hard certain structured benchmarks (such as graph coloring and Hanoi) can be. Thus, the only form of failure observed in these experiments was timeouts, with no recursion depth errors encountered.

4.2 Random 3-SAT Formulas

The experiments on random 3-SAT formulas uf20-91, uf50-218, uf75-325, uf100-430, and uf125-538 benchmarks, where the first number represents the number of variables and the second represents the number of clauses (e.g., uf20-91 means 20 variables and 91 clauses), reveal inconsistency in performance across smaller versus larger instances.

1. For uf20 problems, all solvers managed to find solutions quickly—mostly under 10 ms. DPLL Baseline returned results in approximately 1–8 ms consistently. The frequency and weight-based heuristics added time (20–50 ms) but remained efficient. These results indicate that DPLL is sufficient for small random formulas and that simple heuristics provide little advantage.

2. For uf50, distinctions became more evident. DPLL returned results within 1–10 s, while frequency heuristics needed up to 12 s. The weighted heuristic was consistently faster, often solving instances in ¡ 1 s. The hybrid heuristic was inconsistent—sometimes matching DPLL or performing better, other times running longer or timing out.

3. With the larger UF benchmarks (uf75–uf125), contrasts became sharper, with runtimes and timeouts increasing directly with variable count. Weighted-clause heuristics scaled better than simple frequency counting or other methods. DPLL remained competitive on small formulas but lost efficiency as variables grew. Hybrid methods showed mixed performance, sometimes fast but often unstable.

Taken together, the weighted heuristic scaled best across random instances, DPLL remained strong on smaller ones, and hybrid methods were inconsistent.

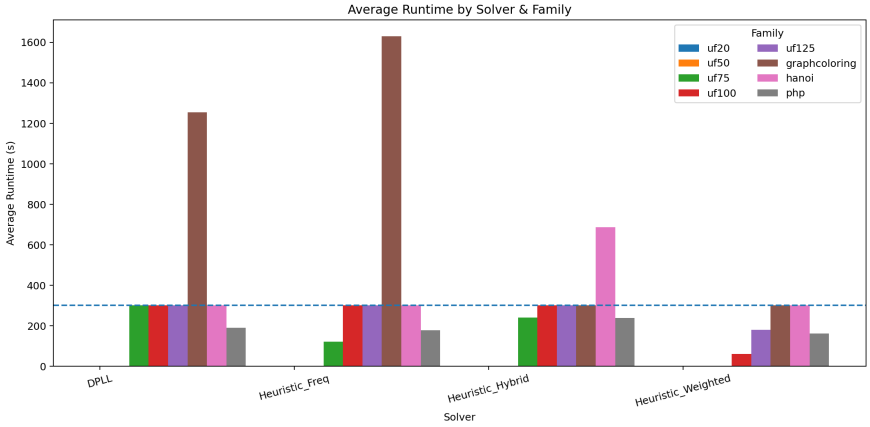

Figure 1: Average runtimes of solvers on random 3-SAT instances (UF20). All solvers managed to find a solution for these smallest instances in nearly instant time (milliseconds). DPLL was very competitive and frequency, weighted, and hybrid heuristics added little overhead or improvement.

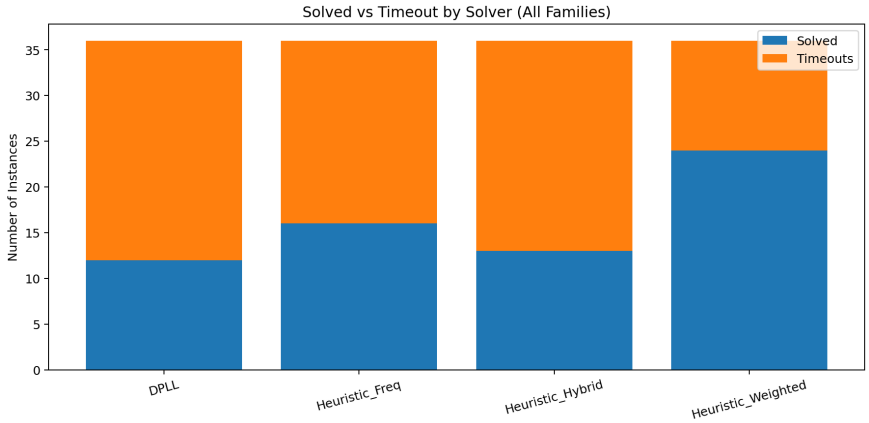

Figure 2: Mean runtime results of all solvers on UF50 random 3-SAT formulas. As averages were produced, patterns of performance emerged: DPLL typically took a few seconds, frequency was slower, weighted was fastest (mostly ¡ 1 s), and hybrid varied widely.

4.3 Hard Problems

The formulated problems exhibited markedly different behavior from 3-SAT.

1. **Pigeonhole Principle:** Larger problems (hole8–hole10) almost always timed out. Smaller problems were more varied: every solver solved hole6 within 1–2 s, and hole7 was solved within 15 s by frequency and weighted heuristics, while DPLL and hybrid were slower. Similar to hole7, hole8 was solved by frequency and weighted at the edge of the 300 s cap (190–265 s), but DPLL and hybrid timed out. These results show it is harder for solvers to prove unsatisfiability than to find a satisfying assignment.

2. **Towers of Hanoi:** Among the hardest problems. For hanoi4 and hanoi5, almost every solver timed out. The deep recursive structure made branching heuristics ineffective.

3. **Graph Coloring:** The most challenging class overall. Many runs saturated the 300 s limit; g250.29 yielded runtimes of thousands of seconds (over 4000 s with DPLL and over 5000 s for frequency). These structured, combinatorial formulas repeatedly pushed solvers to their limits, indicating exponential blow-ups.

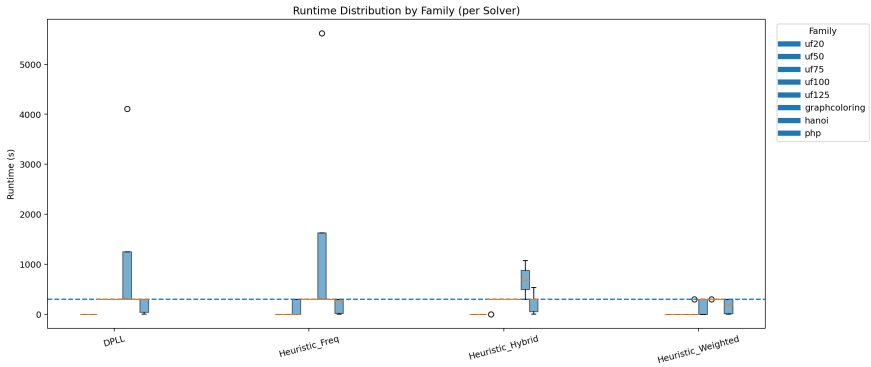

Figure 3: Mean runtimes of all solvers on larger UF benchmarks (UF75–UF125). Weighted heuristics scaled best, often finishing under the 300 s cap, while DPLL, frequency, and hybrid solvers regularly timed out.

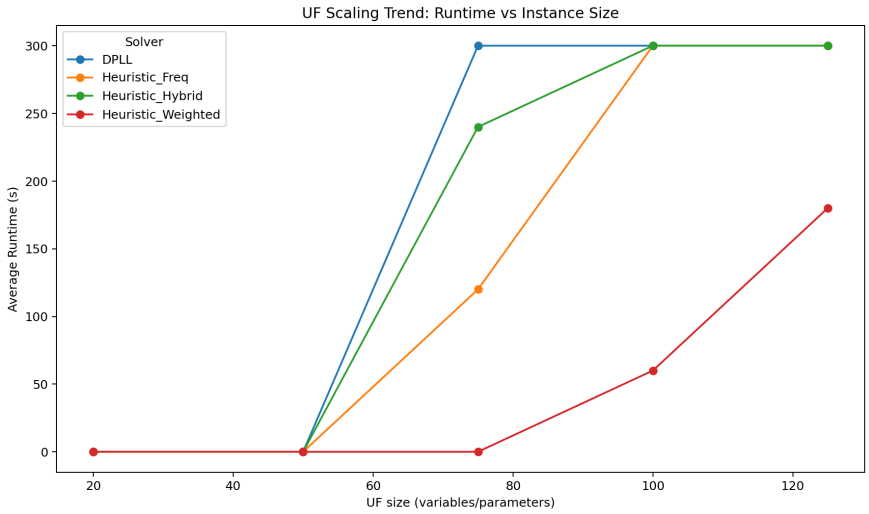

Figure 4: Typical runtimes of the solvers on pigeonhole and Hanoi benchmarks. Pigeonhole encodings stopped solving past the smallest case; Hanoi formulas were the most difficult overall, leading to timeouts even on small instances.

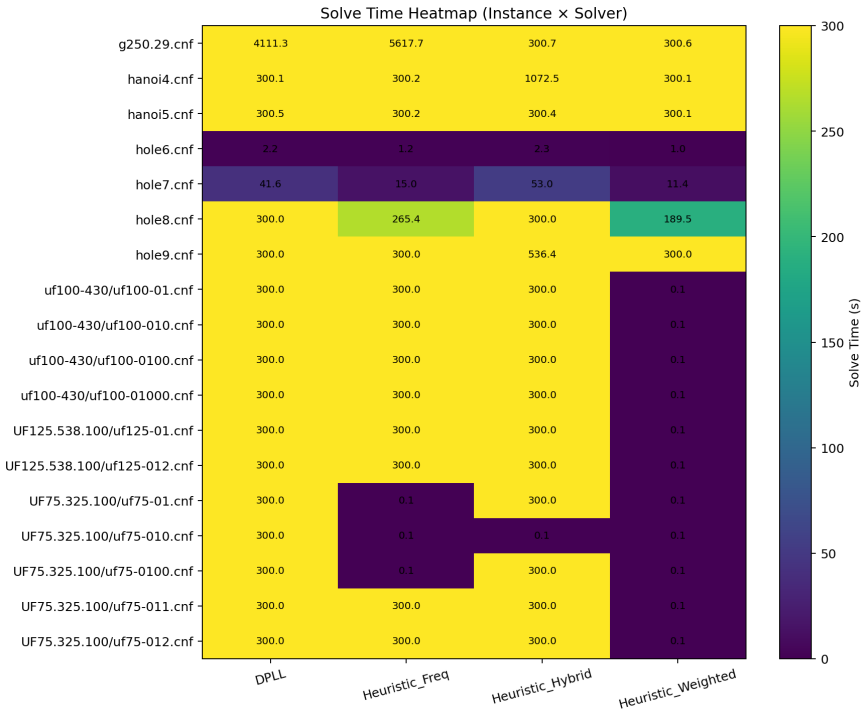

Figure 5: Average runtimes of all solvers on graph-coloring benchmarks. Graph coloring was by far the hardest set; most runs timed out and no solver showed a consistent advantage on these structured instances.

4.4 Empirical Scaling Analysis

We conduct a quantitative runtime analysis on both random and structured SAT instances (pidgeonhole, hanoi, graph coloring). For each solver, an exponential model of the form $T(n) = ae^{bn}$ was fit to the runtime data, where n is the number of variables and b represents the empirical growth rate. A logarithmic scale was then used on the y-axis to better visualize exponential trends.

Random 3-SAT Instances. For the random formulas, the fitted exponents were relatively small: DPLL ($b = 0.144$), Frequency ($b = 0.108$), Weighted ($b = 0.029$), and Hybrid ($b = 0.023$). Low b values correspond to slow exponential growth, indicating that solver runtime increases modestly with the number of variables. This behavior aligns with expectations, as random SAT instances often contain structural randomness that allows early satisfiability pruning and easy variable assignments. Even simple heuristics such as frequency and weighted selection scaled gently, suggesting that these random instances are not inherently hard for the solvers implemented.

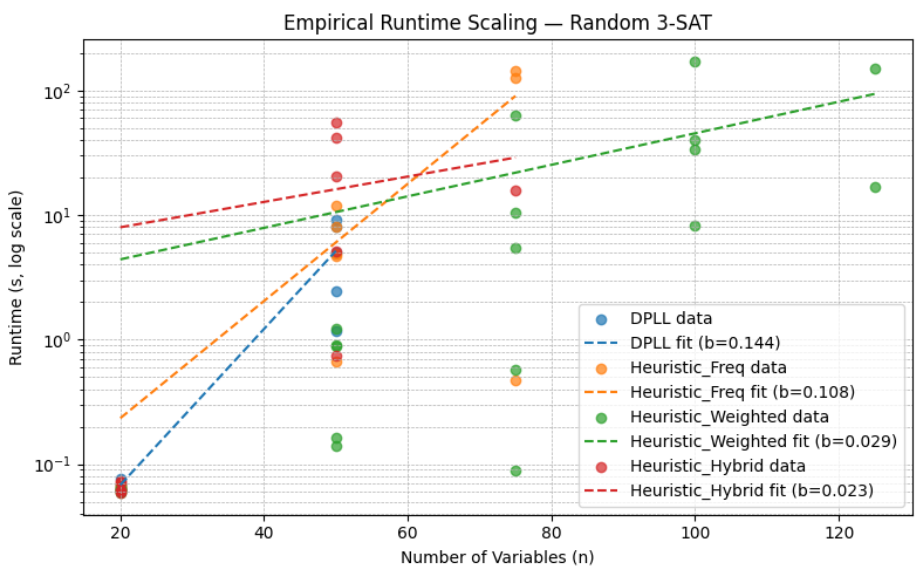

Figure 6: Empirical runtime scaling on random 3-SAT instances. The fitted exponents (b) are small across all solvers, showing gradual scaling with the number of variables. Randomly generated problems often contain enough variability for solvers to exploit simple pruning strategies.

Structured (Mathematical) Instances. The behavior was markedly different for the structured, mathematically encoded benchmarks. Fitted exponents were substantially larger: Frequency ($b = 2.872$) and Weighted ($b = 2.809$). These values are nearly thirty times larger than those observed for the random formulas, revealing exponential blow-up in runtime as

problem size increases. The almost parallel exponential fits in Figure 7 show that while the Weighted heuristic remains marginally faster, both strategies suffer from steep scaling.

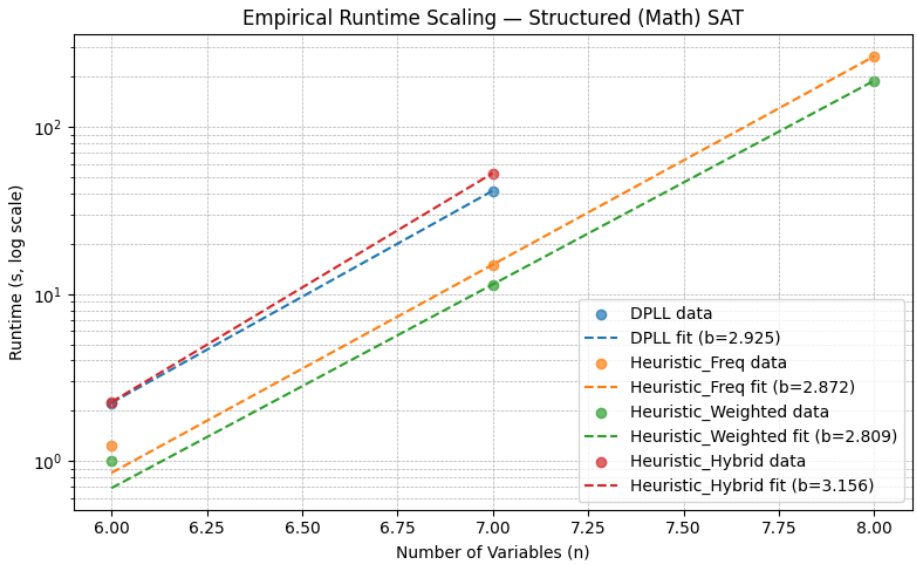

Figure 7: Empirical runtime scaling on structured (mathematical) SAT instances. Both heuristics exhibit steep exponential growth with nearly parallel fitted trends, confirming the inherent difficulty of mathematically structured encodings.

4.5 Predictive Runtime Model

We construct a predictive model for solver runtime as a function of instance size (measured by number of variables). For each solver and benchmark family, runtime $T(n)$ was modeled by the exponential function

$$T(n) = ae^{bn},$$

where a is a scaling constant and b represents the empirical growth rate. Fitting this model to experimental data produced specific-to-solver parameters, allowing direct runtime prediction.

Table 1: Fitted parameters for the exponential runtime model $T(n) = ae^{bn}$.

Solver	**Dataset**	**a**	**b**
DPLL	Random (low-n)	0.000000	0.144
Heuristic_Freq	Random (low-n)	0.000000	0.108
Heuristic_Weighted	Random (low-n)	0.000000	0.029
Heuristic_Hybrid	Random (low-n)	0.000000	0.023
DPLL	Structured (Math)	0.000000	2.925
Heuristic_Freq	Structured (Math)	0.000000	2.872
Heuristic_Weighted	Structured (Math)	0.000000	2.809
Heuristic_Hybrid	Structured (Math)	0.000000	3.156

These models enable quantitative prediction of solver performance. For example, the weighted heuristic on random 3-SAT can be approximated as $T(n) \approx e^{0.029n}$, while the same heuristic on structured SAT follows $T(n) \approx e^{2.809n}$. Substituting a new instance size n into these equations allows us to forecast solver behavior without rerunning experiments.

The predictive power of this framework also demonstrates an important transition in solver performance. For random 3-SAT formulas, runtime scales modestly up to a threshold ($n \approx 70$) before diverging sharply as instances approach the satisfiability phase transition. In contrast, structured "hard" problems (the mathematical encodings) result in consistently steep exponential scaling ($b \approx 2.8$–3.2) across all instance tested.

5 Discussion

By fitting solver runtimes to the exponential function $T(n) = ae^{bn}$, we established a framework that estimates how long a solver will take to complete a problem based on the number of variables. While predictive models for SAT performance have been explored in prior research, this work develops such a model directly from data gathered through our original solver implementations.

The results demonstrate two distinct performances based on the type of SAT formula. For random 3-SAT problems, solver runtime initially increases slowly, remaining manageable across smaller variable counts. However, beyond a critical threshold ($n \approx 70$), the runtimes escalate sharply, reflecting the well-known phase transition in satisfiability. Structured mathematical encodings (the "hard" problems) behave differently, as their exponential scaling appears immediately and consistent. These findings show how different problem structures exhibit fundamentally different growth behaviors under the same solvers.

The predictive model, if further developed with more data for reliability, could be useful in practical application. In fields such as hardware verification, optimization, and artificial intelligence, this model enables practitioners to estimate solver runtime in advance by substituting a given instance size n into the fitted function. That can assist in forecasting computational cost, assessing feasibility, and selecting the most appropriate SAT solver or heuristic strategy.

More broadly, this project demonstrates how empirical modeling can enhance our theoretical understanding. While SAT remains NP-complete in the worst case, the fitted parameters (a, b) provide a quantitative measure of practical difficulty based on algorithmic performance. In this sense, the predictive model contributes to bridging mathematical theory and execution, as well as making time complexity predictable to a meaningful extent.

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/

Fuchs, T., Bach, J., & Iser, M. (2025). Active Learning for SAT Solver Benchmarking. *Journal of Automated Reasoning, 69*, Article 16. https://doi.org/10.1007/s10817-025-09729-6