Unoptimized SAT Solvers in Practice: A Performance Comparison of Resolution, DP, DPLL, and CDCL

Chircan Laurentiu Andrei
Department of Computer Science,
West University Timişoara
`laurentiu.chircan05@e-uvt.ro`

December 24, 2025

Abstract

I present a comparative analysis of four foundational SAT solving algorithms: the Resolution method, Davis–Putnam (DP), Davis–Putnam–Logemann–Loveland (DPLL), and Conflict-Driven Clause Learning (CDCL). While each of these algorithms represents a significant milestone in the search for effective SAT solvers, my focus is on their unoptimized implementations to highlight core algorithmic behaviors without the influence of advanced heuristics or engineering optimizations. I evaluate these solvers on a diverse set of benchmark CNF instances and analyze their performance in terms of runtime. My experiments reveal significant differences in practical efficiency. These results underscore the importance of both algorithmic design and practical considerations in modern SAT solving.

Contents

1 Introduction

The Boolean Satisfiability Problem (SAT) lies at the heart of computational complexity and logic, forming the first problem proven to be NP-complete [5, 10]. SAT asks whether there exists an assignment of truth values to variables that satisfies a given Boolean formula, typically represented in Conjunctive Normal Form (CNF). The problem has extensive applications in fields such as formal verification, artificial intelligence, planning, cryptography, and automated reasoning [4, 11].

Over the decades, numerous algorithms have been developed to tackle SAT, ranging from early symbolic methods to modern, highly optimized search-based solvers. Among the foundational techniques are the Resolution method [15], Davis–Putnam (DP) [7], Davis–Putnam–Logemann–Loveland (DPLL) [6], and Conflict-Driven Clause Learning (CDCL) [13, 18]. Each represents a significant conceptual advancement, with CDCL currently forming the basis of most state-of-the-art SAT solvers [9].

While many studies focus on optimized solver implementations incorporating sophisticated heuristics, preprocessing, and data structures [2], this paper takes a different approach. I implement and analyze unoptimized versions of these four algorithms to provide a clearer understanding of their inherent algorithmic characteristics and comparative strengths. By stripping away engineering enhancements, I aim to assess how each method performs in its most basic form when applied to a common set of benchmark problems [8].

To better understand the SAT problem, consider a simple example: suppose we have three variables—A, B, and C—and a formula composed of the following clauses: (A $\vee$ ¬B), (¬A $\vee$ C), and (¬C $\vee$ B). The question is whether we can assign true or false to each variable in such a way that all the clauses are satisfied. For example, if A is true, B is false, and C is true, each clause evaluates to true, and the formula is satisfiable. This type of reasoning—finding a satisfying assignment or proving none exists—is at the core of SAT solving. As the number of variables and clauses grows, however, the problem becomes exponentially harder [10], highlighting the need for efficient solving algorithms.

By removing layers of engineering and optimization, I aim to isolate and expose the core algorithmic differences that drive performance. This kind of analysis is valuable for multiple reasons. First, it helps demystify the actual contribution of each algorithmic innovation, independent of implementation tricks [12]. Second, it provides an educational baseline for students and researchers interested in building or understanding SAT solvers from the ground up [4]. Lastly, it offers insight into how algorithm design choices affect performance on SAT instances of varying size and structure—a crucial step toward developing more efficient and theoretically grounded solvers.

2 Boolean Satisfiability: Theory and Core Methods

The Boolean Satisfiability Problem (SAT) is the problem of determining whether there exists an assignment of truth values to variables that makes a given Boolean formula evaluate to true.[5] More formally:

Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of Boolean variables. A **literal** is either a variable $x_i \in X$ (a positive literal) or its negation $\neg x_i$ (a negative literal). A **clause** is a disjunction of literals $(l_1 \vee l_2 \vee \ldots \vee l_k)$, and a **CNF formula** φ is a conjunction of clauses $C_1 \wedge C_2 \wedge \ldots \wedge C_m$, where each clause C_i contains a finite number of literals.

A **truth assignment** $\tau : X \to \{0, 1\}$ maps each variable to either true (1) or false (0). A clause is satisfied by τ if at least one of its literals is assigned true under τ; a CNF formula is satisfied if all its clauses are satisfied. The SAT problem asks: *Does there exist an assignment τ such that $\varphi(\tau) = true$?*

Solving SAT is fundamentally a search problem in a combinatorial space of 2^n possible assignments. Various algorithms have been developed to approach this problem from different perspectives. This paper focuses on four classical approaches: **Resolution**, **Davis–Putnam (DP)**, **Davis–Putnam–Logemann–Loveland (DPLL)**, and **Conflict-Driven Clause Learning (CDCL)**. Each represents a milestone in the development of SAT solvers and is grounded in different theoretical principles.[11]

2.1 Resolution Method

The Resolution method is a rule-based, deductive system. It relies on the **resolution rule**, which states that from two clauses $(A \vee x)$ and $(B \vee \neg x)$, one may infer a new clause $(A \vee B)$, called the **resolvent**. This process is repeated until either the empty clause $(\perp)$ is derived—indicating unsatisfiability—or no new clauses can be generated—implying satisfiability.[15]

- **Pros**: Resolution is refutation-complete and forms the basis of many formal proof systems.
- **Cons**: It can be highly inefficient in practice due to combinatorial explosion in the number of derived clauses, especially for satisfiable formulas where no contradiction is found.

2.2 Davis–Putnam (DP) Algorithm

The DP algorithm, introduced in 1960, was one of the first attempts to automate theorem proving using the resolution rule. It eliminates variables one at a time by generating all possible resolvents between clauses containing the variable and its negation, then removing those clauses. This process is repeated recursively on the remaining formula.[7]

- **Pros**: DP simplifies formulas without exploring the space of truth assignments directly. It guarantees termination and is complete.
- **Cons**: The main drawback is the exponential growth in the number of clauses due to full resolution at each step. It also provides no early termination on satisfiable formulas.

2.3 Davis–Putnam–Logemann–Loveland (DPLL) Algorithm

DPLL improves upon DP by replacing resolution-based elimination with a **backtracking search** combined with **unit propagation** and **pure literal elimination**. It recursively assigns truth values to variables and simplifies the formula at each step. If a conflict arises, the algorithm backtracks and tries alternative assignments.[6] [12]

- **Pros**: DPLL can often solve practical instances efficiently due to pruning and simplification. It avoids the clause explosion of DP by exploring assignments instead of exhaustive resolution.
- **Cons**: In worst-case scenarios, DPLL still explores an exponential number of assignments and may revisit similar states multiple times.

2.4 Conflict-Driven Clause Learning (CDCL)

CDCL is a modern enhancement of DPLL that introduces **clause learning**, **non-chronological backtracking** (also known as backjumping), and **restarts**. When a conflict is detected during propagation, CDCL analyzes the implication graph to learn a new clause that prevents the same conflict from recurring. This new clause is added to the formula, allowing the solver to prune large portions of the search space.[13] [9]

- **Pros**: CDCL dramatically improves solver performance in practice, particularly on industrial instances. Its ability to learn from conflicts gives it a powerful advantage in avoiding redundant search paths.
- **Cons**: While CDCL is highly effective, its theoretical guarantees are similar to DPLL, and its performance still depends heavily on heuristics and learned clause management. In an unoptimized setting, the cost of conflict analysis and clause learning can outweigh the benefits for small or simple instances.

3 2SAT and 3SAT: Special Cases of SAT

While SAT in its general form is NP-complete, certain subclasses of the problem exhibit markedly different computational properties. Two important special cases are **2SAT** and **3SAT**, where the CNF formula is restricted such that each clause contains exactly two or three literals, respectively.

2SAT: A Polynomial-Time Solvable Subclass

2SAT refers to SAT instances in which every clause contains exactly two literals, such as $(x \lor \neg y)$. Unlike general SAT, 2SAT is solvable in **polynomial time**. This tractability arises from the structure of its implication graph: every 2SAT formula can be transformed into a directed graph where a clause like $(x \lor y)$ becomes two implications: $(\neg x \to y)$ and $(\neg y \to x)$. The formula is satisfiable if and only if *no variable and its negation lie in the same strongly connected component* [1].

Efficient 2SAT solvers exploit this property using algorithms like *Kosaraju's* or *Tarjan's algorithm* [17], both of which run in linear time relative to the number of clauses and variables. As shown in Section 5.1, even unoptimized implementations of DPLL and CDCL perform remarkably well on 2SAT, although CDCL's advantage is more muted due to the solvability of 2SAT via simpler graph-based methods.

3SAT: The Benchmark of Hardness

In contrast, **3SAT**, where every clause has exactly three literals (e.g., $(x \lor \neg y \lor z)$), retains the full computational hardness of SAT and is a canonical **NP-complete** problem. It was historically the first problem proven to be NP-complete (via Cook's theorem) [5], and many reductions in complexity theory use 3SAT as a starting point.

Due to its expressive power and hardness, 3SAT is frequently used as a benchmark in both theoretical research and practical solver evaluations. Unlike 2SAT, no known polynomial-time algorithm exists for solving 3SAT unless $\mathsf{P} = \mathsf{NP}$. Thus, 3SAT instances better showcase the strengths and weaknesses of SAT algorithms, especially in regard to pruning strategies, clause learning, and backtracking efficiency.

As shown in Section 5.2, the performance difference between DPLL and CDCL becomes far more pronounced on 3SAT instances. CDCL's ability to learn from conflicts and perform non-chronological backtracking allows it to avoid redundant explorations that slow down DPLL substantially.

4 Modeling and Implementation

Now, we will explore the implementations of the algorithms mentioned above. C++ 11 was chosen for it's simplicity, easy access to relevant libraries and speed. Python would have also been a good choice but due to its interpreted nature i preferred to go with C++. Also this choice aligns with the universal choice of language when it comes to SAT solver.

Taking into account that the main objective was to analyze and compare the performance of these algorithms, i needed a way to process multiple problems and report the necessary time to come to a conclusion. The answer was another standard in SAT problems, the DIMACS format. Thus all the algorithms have an implementation of a batch processing method in which we just take all the CNF files with SAT problems stored in DIMACS format and iterate through them. We are also using the `<chrono>` library to measure execution time.[16]

4.1 CNF Formula Representation and the DIMACS format

SAT problems are modeled in Conjunctive Normal Form (CNF) and then represented in DIMACS format, where each formula is a conjunction of clauses, and each clause is a disjunction of literals.[8] In our implementation:

- Variables are represented as integers: a positive integer x denotes the literal x, and a negative integer $-x$ denotes $\neg x$.
- Clauses are stored as lists of integers.
- A formula is a list of such clauses.

For example, the CNF formula:

$$(A \vee \neg B) \wedge (\neg A \vee C) \wedge (\neg C \vee B)$$

is encoded as:

$$[[1, -2], [-1, 3], [-3, 2]]$$

where $A = 1$, $B = 2$, and $C = 3$.

4.2 Implementation Overview

Each algorithm was implemented as a standalone C++ console program.

Resolution. The resolution engine iteratively applies the resolution rule to all resolvable clause pairs until either an empty clause is derived or no new clauses are produced. Clauses are stored in a set for efficient lookup and redundancy avoidance.

Davis–Putnam (DP). The DP algorithm recursively eliminates variables by full resolution on all clauses containing a variable and its negation. At each step, the affected clauses are replaced by their resolvents, and the original clauses are removed.

DPLL. The DPLL algorithm was implemented using recursive backtracking. It includes unit propagation and pure literal elimination as simplification techniques. The solver branches on unassigned variables and backtracks upon encountering conflicts.

CDCL. CDCL was built upon the DPLL framework with additional mechanisms:

- Implication graph construction
- Conflict analysis using the First Unique Implication Point (UIP)
- Learned clause addition
- Non-chronological backtracking (backjumping)

Restarts and variable decision heuristics (e.g., VSIDS) were *not* implemented to preserve the unoptimized nature of the study. [14]

4.3 Test Environment

- **Language:** C++ 11
- **Hardware:** AMD Ryzen 7 CPU @ 3.10GHz, 24GB RAM
- **Operating System:** Windows 11 24H2
- **Time Measurement:** `std::chrono::high_resolution_clock`.

4.4 How to use the implementations

Each algorithm has its own implementation that can be used independently. When run, it reads all the CNF files from the designated folder and outputs the result in a result.txt file in the following format: **filename: SAT/UNSAT in x.xxx ms**. There are also outputs in the console with the same format. Further explanations are provided in the GitHub repository.

5 Empirical Analysis of SAT Solvers: The Numbers

To evaluate the practical performance of the four unoptimized SAT solvers—Resolution, Davis–Putnam (DP), Davis–Putnam–Logemann–Loveland (DPLL), and Conflict-Driven Clause Learning (CDCL)—I conducted a series of experiments on a benchmark set of 60 2SAT problems and 40 3SAT problems expressed in the DIMACS CNF format. These problems were chosen to reflect a mix of satisfiable and unsatisfiable instances with varying numbers of variables and clauses, ranging from 5 to 850 variables and 10 to 8500 clauses.

All solvers were tested on the same machine under identical conditions to ensure consistency. Execution time was measured using the C++ `<chrono>` library, and only solving time was recorded—excluding file parsing and I/O operations.

Table 1: Solving times (in milliseconds) for the SAT solvers on selected 2SAT instances.

Instance	**Resolution**	**DP**	**DPLL**	**CDCL**
2sat_10_40_1_unsat	7.15	0.14	0.30	0.04
2sat_20_40_sat	84.95	0.56	0.05	0.02
2sat_20_80_1_unsat	65.76	0.50	0.50	0.03
2sat_20_160_2_unsat	75.32	0.82	0.40	0.04
2sat_20_320_unsat	10.98	1.58	1.93	0.04
2sat_40_560_unsat	265.77	3.31	2.57	0.24
2sat_40_560_2_unsat	364.92	3.03	2.53	0.26
2sat_60_750_unsat	485.78	5.38	4.20	0.13
2sat_70_850_unsat	164.23	6.95	3.60	0.19
2sat_80_1400_unsat	7530.00	17.58	9.57	0.29
2sat_160_2500_1_unsat	1239.47	42.28	17.26	0.71
2sat_160_2500_2_unsat	20839.45	66.08	15.69	0.44
2sat_180_2800_unsat	5827.18	54.09	18.26	0.75
2sat_250_4050_unsat	68406.07	187.53	39.85	0.88
2sat_300_4540_unsat	349884.35	154.26	69.48	1.28
2sat_450_7540_unsat	699890.39	322.30	61.99	3.97
2sat_550_9540_unsat	226986.12	838.76	99.97	0.08
2sat_850_1240_unsat	113721.257	80.12	176.13	4.53

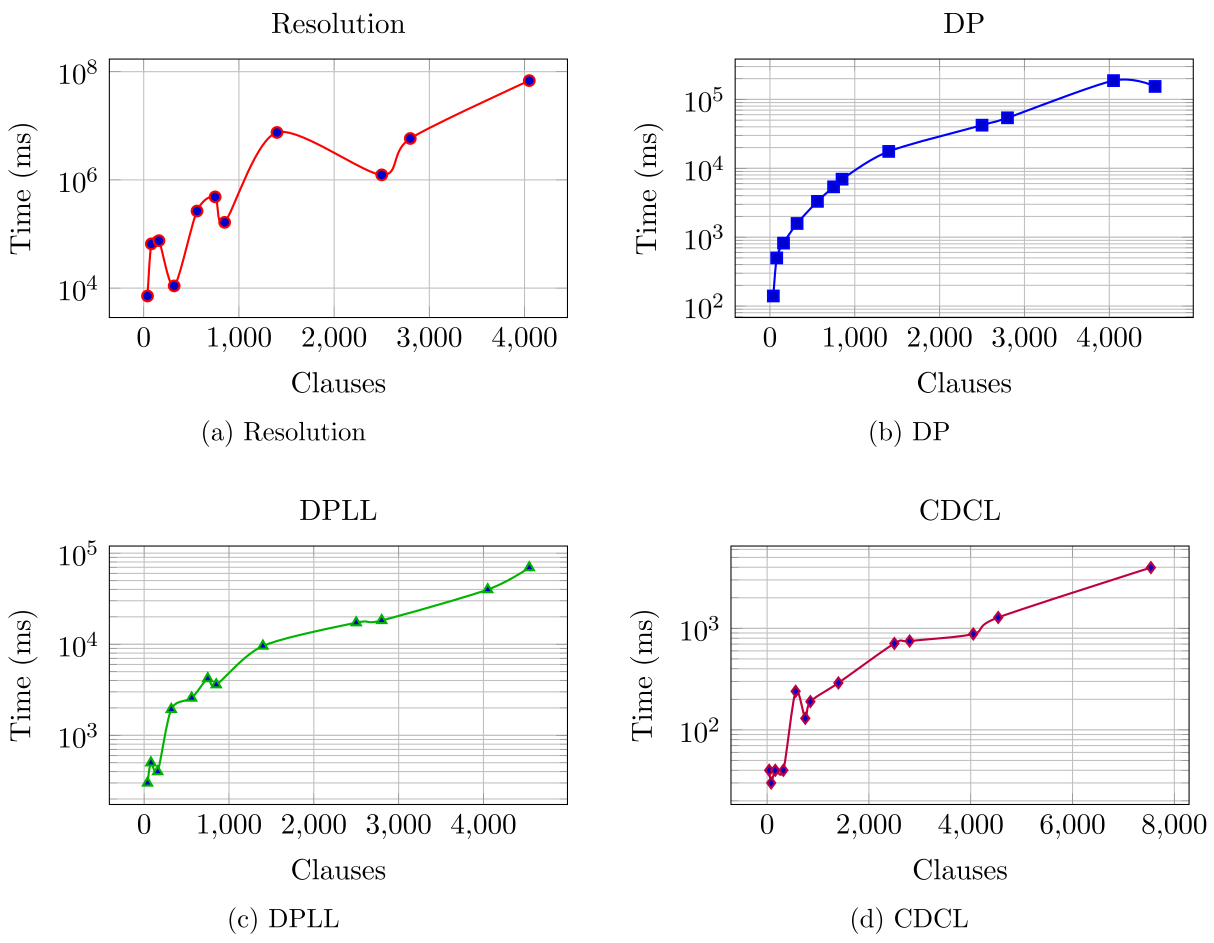
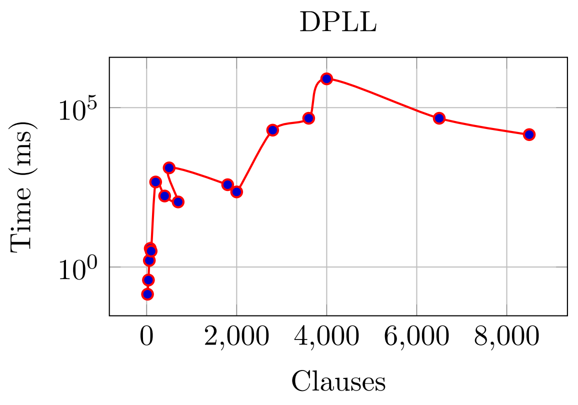


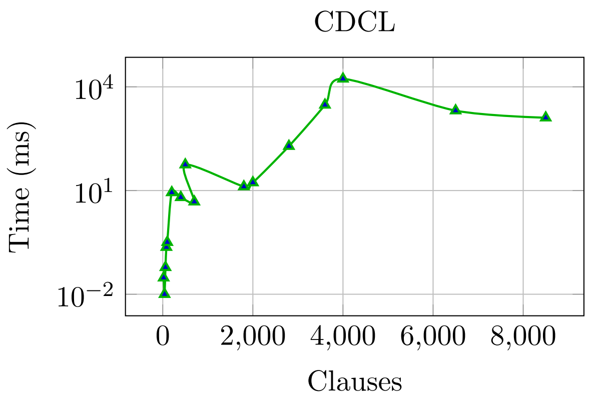
Figure 1: Solving Time vs Clauses (Log Scale, Time in Milliseconds)

Table 2: Solving times (in milliseconds) for DPLL and CDCL solvers on selected 3SAT instances.

Instance	**DPLL**	**CDCL**
3sat_10_20_sat	0.14	0.03
3sat_10_40_unsat	0.39	0.01
3sat_10_60_unsat	1.61	0.06
3sat_15_80_unsat	3.82	0.23
3sat_50_100_sat	3.15	0.32
3sat_50_200_unsat	465.59	8.79
3sat_50_400_unsat	169.49	6.35
3sat_50_700_unsat	110.26	4.74
3sat_80_500_unsat	1290.92	58.626
3sat_100_1800_unsat	382.10	13.26
3sat_100_2000_unsat	227.38	17.08
3sat_180_2800_unsat	19669.43	192.10
3sat_260_3600_unsat	46423.67	3000.96
3sat_300_4000_unsat	800157.24	17019.50
3sat_300_6500_1_unsat	46428.49	2049.92
3sat_300_8500_2_unsat	14241.75	1279.72



(a) DPLL



(b) CDCL

Figure 2: Solving Time vs Clauses (Log Scale, Time in Milliseconds)

5.1 Performance Comparison Summary (2SAT Instances)

Table 1 summarizes the solving times (in milliseconds) of four SAT solvers: Resolution, Davis-Putnam (DP), DPLL and CDCL, on 2SAT instances with increasing variables and clause counts.

CDCL emerged as the fastest solver across all benchmarks, often solving instances in under 0.1 milliseconds, even as instance size increased.[3] DPLL was consistently slower than CDCL but maintained millisecond-level solving times. The Davis–Putnam algorithm showed reasonable scalability, but was consistently outperformed by DPLL and CDCL.

In contrast, the Resolution-based solver demonstrated extremely poor scalability, with solving times reaching hundreds of milliseconds and even seconds for larger formulas. For instance, `2sat_450_7540_unsat` took over 699890 milliseconds with Resolution, while CDCL solved it in just under 4 milliseconds—a speedup of over $150,000\times$.

Overall, these results highlight the practical superiority of conflict-driven clause learning (CDCL) and the importance of clause learning and efficient backtracking for SAT solving in modern contexts, even for restricted CNF classes like 2SAT.

5.2 DPLL vs CDCL on 3SAT Benchmarks.

Table 2 presents a comparison of solving times between the DPLL and CDCL solvers on standard 3SAT instances. CDCL consistently outperformed DPLL. DPLL was approximately 5× to 50× slower than CDCL, and the performance gap gradually increased with input size.

For instance, on `3sat_300_4000_unsat`, CDCL completed in 17019.50 ms compared to DPLL's 800157.24 ms. Although both solvers scale reasonably well on this benchmark, CDCL's advantages in clause learning, non-chronological backtracking, and watched literal propagation clearly contribute to its superior efficiency.

These results reinforce CDCL's status as the state-of-the-art algorithm for practical SAT solving and demonstrate its effectiveness even on moderately sized 3SAT problems where DPLL still performs competitively.

6 From Resolution to CDCL

The development of SAT solvers has evolved through a series of fundamental algorithmic breakthroughs, beginning with symbolic methods and culminating in the state-of-the-art CDCL solvers.

One of the earliest theoretical frameworks is the Resolution method introduced by Robinson [15], which provides a complete refutation system but suffers from severe performance limitations in practice due to clause explosion.

The Davis–Putnam (DP) procedure [7] introduced a variable elimination strategy that significantly influenced later work. However, it also encounters scalability issues due to the necessity of exhaustive resolution.

A major improvement came with the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [6], which combines backtracking search with simplification strategies like unit propagation and pure literal elimination. It remains a foundational framework upon which modern solvers are built.

The CDCL algorithm [13, 18] extends DPLL with conflict-driven learning, non-chronological backtracking, and clause database management, enabling remarkable performance gains. These techniques are discussed in detail in comprehensive surveys such as Biere et al. [4].

While most research emphasizes highly optimized implementations with advanced heuristics (e.g., VSIDS, clause deletion strategies), this paper aligns with educational and analytical efforts to examine solver behavior in unoptimized conditions. Such approaches help isolate pure algorithmic performance, as discussed by Bayardo and Schrag [2].

Benchmarking efforts such as the DIMACS challenge [8] provide standardized problem sets that are instrumental in evaluating solver performance across methodologies.

Finally, foundational complexity results by Cook [5] and formal problem classifications by Garey and Johnson [10] frame SAT within the broader context of NP-completeness and computational hardness.

7 Conclusions

This work presented basic SAT solving methods, with the aim of understanding the fundamental mechanisms behind modern SAT solving. The solvers were tested against benchmark instances in DIMACS CNF format, providing insight into both the strengths and limitations of a simplified SAT solving implementation.

Future Work

Several extensions are proposed to address these open issues:

- Implementing dynamic restart policies (e.g., Luby sequence or geometric backoff) to improve convergence on hard problems.
- Incorporating adaptive heuristics such as EVSIDS for smarter decision-making.
- Adding clause activity metrics and garbage collection to prune the clause database.
- Parallelizing unit propagation or clause learning could be explored for performance gains.

Furthermore, integration with a GUI or visualization tool could support educational use cases and better explain the solver's decision process.

Overall, the project provided a solid foundation in understanding SAT solving, highlighting the complexity and nuance behind state-of-the-art solvers while opening avenues for further exploration and optimization.

References

[1] Bengt Aspvall, Michael F. Plass, and Robert E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.

[2] Roberto J Bayardo Jr and Robert C Schrag. Using csp look-back techniques to solve real-world sat instances. *AAAI/IAAI*, pages 203–208, 1997.

[3] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[4] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Handbook of satisfiability. *Frontiers in Artificial Intelligence and Applications*, 185, 2009.

[5] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing (STOC)*, pages 151–158. ACM, 1971.

[6] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[7] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[8] DIMACS Challenge. Satisfiability: Suggested format. `http://dimacs.rutgers.edu/Challenges/Satisfiability/`, 1993. Center for Discrete Mathematics and Theoretical Computer Science.

[9] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT 2003*, pages 502–518. Springer, 2003.

[10] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[11] Chu Min Li and Felip Manyà. Maxsat, hard and soft constraints. *Handbook of Satisfiability*, 185:613–631, 2009.

[12] Donald Loveland. Automated theorem proving: A logical basis. *North-Holland*, 1978.

[13] João P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[14] Matthew W. Moskewicz et al. Chaff: Engineering an efficient sat solver. In *DAC 2001*, pages 530–535, 2001.

[15] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[16] SAT Competition Organizers. Sat competition. `http://satcompetition.org`, ongoing.

[17] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[18] Lintao Zhang and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. *International Conference on Computer Aided Design*, pages 279–285, 2001.