Study on SAT Solvers

Stephanie Matasaru
Department of Computer Science,
West University Timișoara
stephanie.matasaru05@e-uvt.ro

Abstract

Satisfiability (SAT) is a fundamental problem in computer science and logic. This paper presents the implementation in Python of three classical SAT-solving algorithms: Resolution, Davis–Putnam (DP), and DPLL. We tested the algorithms on batches of systems of clauses with different sizes. Based on the results presented in the paper, we identified the weaknesses of the methods and where they could be improved.

Last but not least, some optimizations are highlighted in one of the leading sequential SAT solvers as of 2024. The work aims to provide insight into the evolution of SAT solving.

Keywords: SAT, Resolution, DP, DPLL

Contents

1 Introduction

The Boolean Satisfiability Problem (SAT) is one of the most significant problems in computer science theory. SAT examines whether a Boolean formula becomes true when its variables receive specific truth assignments. SAT was the first proven NP-complete problem, as shown by Cook in 1971 [5], and it remains essential to complexity research today. Its impact extends beyond theory to practical domains such as artificial intelligence, reasoning systems, and hardware verification.

To better understand what SAT solving involves, consider two simple Boolean formulas:

Example 1 (Satisfiable).

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

This formula is satisfiable. One possible assignment is: A = true, B = true, C = false. The evaluation is shown below:

A	B	C	$\neg A \vee B$	$\neg B \vee C$	$\neg C$	A	Formula
T	T	F	T	F	T	T	T

Example 2 (Unsatisfiable).

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

This formula is unsatisfiable — there is no way to assign truth values to satisfy all clauses. The following truth table presents all possible truth value assignments:

A	B	C	$\neg A$	$\neg B$	$\neg C$	$A \vee B$	$\neg A \vee C$	$\neg B \wedge \neg C$	Formula
F	F	F	T	T	T	F	T	T	F
F	F	T	T	T	F	F	T	F	F
F	T	F	T	F	T	T	T	F	F
F	T	T	T	F	F	T	T	F	F
T	F	F	F	T	T	T	F	T	F
T	F	T	F	T	F	T	T	F	F
T	T	F	F	F	T	T	F	F	F
T	T	T	F	F	F	T	T	F	F

Small formulas like these can be manually evaluated, but the number of possible assignments grows exponentially with the number of variables, making automated, efficient solving essential.

The early SAT solving methods were based on logical inference. The most well-known of these is the **resolution** method, which tries to derive a contradiction by repeatedly applying inference rules to formula clauses. While resolution is complete, it is often too slow in practice because of the large number of clauses it can generate.

To improve performance, in 1960 Davis and Putnam came up with the Davis–Putnam algorithm (DP). Instead of pure resolution, DP methodically eliminated variables by resolving clauses containing them, then simplified the resulting formula. Although DP improved upon earlier methods, it still has limitations, especially when dealing with large instances.

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm brought a significant improvement through a fundamentally different approach. Instead of resolution, DPLL uses backtracking alongside two other techniques: *unit propagation*, which assigns values to variables that must be true, and *pure literal elimination*, which assigns values to literals that appear with only one polarity in the formula. These changes make DPLL much more efficient than previous methods.

Current SAT solvers still face major difficulties when dealing with large or complex formulas. Modern techniques like clause learning, conflict-driven clause learning (CDCL), and heuristic-based variable selection have made solvers far more powerful, they are not guaranteed to run efficiently on all instances.

This paper takes a theoretical and practical approach, starting with an exploration of the SAT problem and early algorithmic techniques. It then moves on to the implementation and evaluation of classical methods such as resolution, the DP algorithm, and DPLL, and then a detailed analysis of their strengths and limitations. For those interested in solver performance and benchmarking, the experimental section provides valuable insights. The paper then shifts to modern SAT solving techniques, including conflict-driven clause learning (CDCL) and heuristic-based variable selection, and includes a case study of the Kissat solver. The goal is to understand how modern methods build on classical foundations, why they perform better in practice.

This work is the result of independent study and implementation. All algorithms were implemented and tested by the author, and sources used for background information or comparative analysis are cited appropriately throughout the document.

2 Formal Description of the Problem and Solution

The Boolean Satisfiability Problem (SAT) is defined as follows: given a propositional formula φ built using Boolean variables and logical connectives ($\neg$, $\wedge$, $\vee$), determine whether there exists a valuation (assignment of truth values) such that φ evaluates to true.

Formally, let $\mathcal{L}$ be the propositional language generated over a finite set of propositional variables $\{P_1, P_2, \ldots, P_n\}$. A model is a function $v : \{P_1, \ldots, P_n\} \to \{\text{true}, \text{false}\}$. A formula $\varphi \in \mathcal{L}$ is said to be satisfiable if there exists a model v such that $v(\varphi) = \text{true}$.

This definition establishes SAT as a decision problem:

$$\text{SAT} = \{\varphi \mid \varphi \text{ is satisfiable}\}.$$

SAT was the first problem proven to be NP-complete, as shown by Cook in 1971 [5]. This result implies that if a polynomial-time algorithm for SAT is found, it would solve all NP problems efficiently.

Conjunctive Normal Form and Clause Sets

A formula is in *Conjunctive Normal Form* (CNF) if it is a conjunction of one or more disjunctions of literals:

$$\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_k, \quad \text{where } C_i = \ell_1 \vee \ell_2 \vee \cdots \vee \ell_m,$$

and each ℓ_i is either a variable P_j or its negation $\neg P_j$.

Every propositional formula can be converted into an equisatisfiable CNF formula using standard logical equivalences and auxiliary variables [6].

Once in CNF, a SAT instance is often represented as a set of *clauses*, where each clause is a set of literals. For example, the formula $(A \vee B) \wedge (\neg A \vee C) \wedge (\neg B)$ becomes the clause set:

$$\mathcal{C} = \{\{A, B\}, \{\neg A, C\}, \{\neg B\}\}.$$

2.1 Resolution Method

The resolution method is an inference rule applied to clause sets. Given two clauses containing complementary literals, resolution derives a new clause (the resolvent) by eliminating the complementary pair.

Formally, the resolution rule is:

$$\frac{C_1 = A \cup \{p\}, \quad C_2 = B \cup \{\neg p\}}{C = A \cup B},$$

where C_1, C_2 are clauses and p is a propositional variable. If $\emptyset$ (the empty clause) can be derived, the formula is unsatisfiable [6].

The algorithmic approach to applying resolution is illustrated below. This version is adapted from the method described by Crăciun [6].

Algorithm 1 Resolution-Based SAT Decision Procedure

Input: Clause set $\mathcal{K}$
Output: "Satisfiable" or "Not Satisfiable"
$\mathcal{K}' \leftarrow \mathcal{K}$
while there exists a resolvent C of two clauses in $\mathcal{K}'$ such that $C \notin \mathcal{K}'$ **do**
    if $C = \emptyset$ **then**
        return "Not Satisfiable"
    else
        $\mathcal{K}' \leftarrow \mathcal{K}' \cup \{C\}$
    end if
end while
return "Satisfiable"

Example. We begin with $\mathcal{K} = \{\{A, B\}, \{\neg A\}, \{\neg B, C\}, \{\neg C\}\}$.

- Resolve $\{A, B\}$ and $\{\neg A\}$ to get $\{B\}$
- Resolve $\{B\}$ and $\{\neg B, C\}$ to get $\{C\}$
- Resolve $\{C\}$ and $\{\neg C\}$ to get $\emptyset$

Thus, the empty clause is derived, and the formula is unsatisfiable.

While this approach is a complete method for determining unsatisfiability in propositional logic, resolution can become inefficient due to the explosion in the number of resolvents generated for larger clause sets. In later sections, we will analyze its behavior and compare it to more efficient approaches.

2.2 The Davis–Putnam Algorithm (DP)

The Davis–Putnam (DP) algorithm improves upon pure resolution by successively eliminating variables. For each variable p, it identifies all clauses where p appears positively and negatively, generates all possible resolvents from such pairs, and then removes every clause containing p or $\neg p$. This process continues until either:

- the empty clause is derived $\Rightarrow$ the formula is unsatisfiable, or

- the clause set becomes empty $\Rightarrow$ the formula is satisfiable.

Algorithm 2 Davis–Putnam SAT Decision Procedure

Input: Clause set $\mathcal{K}$
Output: "Satisfiable" or "Not Satisfiable"
while $\mathcal{K} \neq \emptyset$ **do**
    if $\emptyset \in \mathcal{K}$ **then**
        return "Not Satisfiable"
    end if
    Choose a variable p appearing in $\mathcal{K}$
    Let $\mathcal{K}_p^+$ be clauses with p, $\mathcal{K}_p^-$ be clauses with $\neg p$
    Compute all resolvents between $\mathcal{K}_p^+$ and $\mathcal{K}_p^-$
    $\mathcal{K} \leftarrow (\mathcal{K} \cup \text{resolvents}) \setminus \{\text{clauses with } p \text{ or } \neg p\}$
end while
return "Satisfiable"

Example. Let us apply DP to the same clause set $\mathcal{K} = \{\{A, B\}, \{\neg A\}, \{\neg B, C\}, \{\neg C\}\}$.

- Choose variable A:
  - $\mathcal{K}_A^+ = \{\{A, B\}\}$, $\mathcal{K}_A^- = \{\{\neg A\}\}$
  - Resolvent: $\{B\}$
  - Remove all clauses with A or $\neg A$
  - New clause set: $\{\{B\}, \{\neg B, C\}, \{\neg C\}\}$
- Choose B:
  - Resolvent of $\{B\}$ and $\{\neg B, C\}$: $\{C\}$
  - Remove all clauses with B or $\neg B$
  - New clause set: $\{\{C\}, \{\neg C\}\}$
- Resolvent of $\{C\}$ and $\{\neg C\}$: $\emptyset \Rightarrow$ unsatisfiable

Although the DP algorithm has exponential worst-case complexity, it offers a more systematic alternative to brute-force search [8].

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

The DPLL algorithm combines backtracking search with two inference techniques:

- **Unit propagation:** If a clause contains only one literal (a unit clause), that literal must be true in any satisfying assignment.
- **Pure literal elimination:** If a literal appears in the formula with only one polarity (e.g., only p and not $\neg p$), then it can be safely assigned true.

At each step, DPLL chooses a variable, assigns it a truth value, simplifies the clause set using unit propagation and pure literal elimination, and recursively explores both branches. Backtracking occurs when a conflict is encountered.

DPLL is the foundation of modern SAT solvers and significantly more efficient than naive search or resolution alone [7].

Algorithm 3 DPLL SAT Decision Procedure

Input: Clause set $\mathcal{K}$
Output: "Satisfiable" or "Not Satisfiable"
Simplify $\mathcal{K}$ using unit propagation and pure literal elimination
if $\emptyset \in \mathcal{K}$ **then**
    return "Not Satisfiable"
else if $\mathcal{K} = \emptyset$ **then**
    return "Satisfiable"
else
    Choose a variable p
    return DPLL($\mathcal{K}[p \mapsto \text{true}]$) or DPLL($\mathcal{K}[p \mapsto \text{false}]$)
end if

Example. We apply DPLL recursively to $\mathcal{K} = \{\{A, B\}, \{\neg A\}, \{\neg B, C\}, \{\neg C\}\}$.

- Unit clause $\{\neg A\} \Rightarrow$ assign $A :=$ false
- Simplify:
$$\mathcal{K}' = \{\{B\}, \{\neg B, C\}, \{\neg C\}\}$$
- Unit clause $\{B\} \Rightarrow$ assign $B :=$ true
- Simplify:
$$\mathcal{K}'' = \{\{C\}, \{\neg C\}\}$$
- Conflict $\Rightarrow$ backtrack; try $B :=$ false $\Rightarrow \mathcal{K}$ has empty clause $\Rightarrow$ unsatisfiable

3 Model and Implementation of the Problem and Solution

Having formally introduced the Boolean Satisfiability Problem and analyzed classical solving algorithms, we now focus on their computational realization. This section details the internal data representations, implementation strategies, and system architecture. It also provides usage instructions and outlines how users can interact with the SAT solver.

3.1 Computational Representation

For implementation purposes, SAT instances are represented as follows:

- Variables are encoded as positive integers $(1, 2, \ldots, n)$.
- Negated variables are represented as negative integers (e.g., $\neg A$ is -1).
- Each clause is stored as a set of integers.

- The overall formula is a collection (list or set) of such clauses.

This representation is compatible with the DIMACS CNF format, facilitating input/output interoperability and ease of integration with external tools.

3.2 Implementation Overview

We have implemented the three classical SAT solving algorithms discussed in Section 2. The programs are written in Python. The source code and the examples used as a benchmark are available at `https://github.com/stephanie-matasaru/SAT-Paper`.

The code represents the pseudo-code without any other optimizations. The function that computes the resolvent is the same for both `resolution.py` and `dp.py`. In the same manner, the functions that treat unit propagation and pure literals are the same for `dp.py` and `dpll.py`.

The solver accepts input files in standard DIMACS CNF format and provides a command-line interface for ease of use:

```
$ python [resolution|dp|dpll].py <example.cnf>
```

The output includes:

- The satisfiability status (SAT/UNSAT)
- Timing information
- Number of clauses in the final system (only for Resolution and DP)

4 Case Studies and Experimental Evaluation

4.1 Experimental Setup

For benchmark we used batches of random formulas generated with `cnfgen`. [11]

Each batch contains systems with `n` variables and 4.5 times as many clauses. Each clause contains 3 variables or negated variables. Some of the systems are satisfiable and some are unsatisfiable. We computed the average execution time for each batch. For the large systems, we used 10 or even 5 examples because they take a very long time; for the smaller ones, we used 100 example files.

4.2 Comparative Analysis

We have compared all 3 algorithms on the batches with smaller examples of 5 to 8 variables. The results are shown in Table 1 and Figure 1.

Table 1: Runtime comparison (seconds) for Resolution, DP, and DPLL on smaller SAT instances

Algorithm	**5 / 23**	**6 / 27**	**7 / 32**	**8 / 36**
Resolution	0.032800	0.764252	19.361224	565.595540
DP	0.022336	0.473387	12.976847	347.544805
DPLL	0.000026	0.000033	0.000044	0.000059

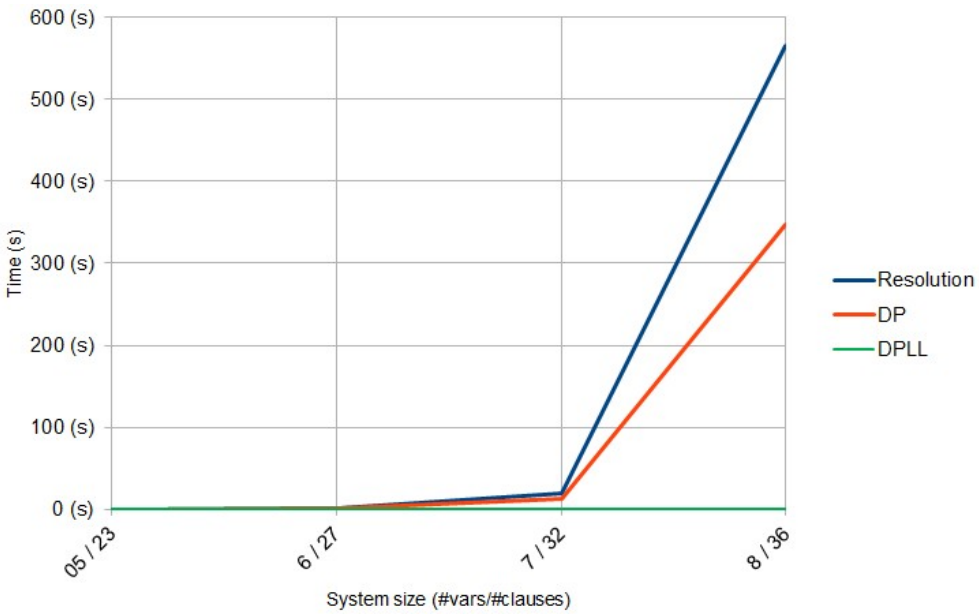

Figure 1: Comparison of solver runtime on benchmark formulas

Table 2: Clause count and runtime comparison for Resolution and DP (uf5-23 and uf6-27)

Example	**Status**	**Resolution (clauses)**	**Resolution (time)**	**DP (clauses)**	**DP (time)**
examples/uf5-23/uf5-0001.cnf	SAT	180	0.049838	0	0.024127
examples/uf5-23/uf5-0002.cnf	SAT	153	0.033216	42	0.016771
examples/uf5-23/uf5-0003.cnf	UNSAT	210	0.033285	210	0.032406
examples/uf5-23/uf5-0004.cnf	UNSAT	210	0.030288	210	0.030518
examples/uf5-23/uf5-0005.cnf	UNSAT	210	0.030904	210	0.030323
examples/uf5-23/uf5-0006.cnf	UNSAT	210	0.031608	210	0.031386
examples/uf5-23/uf5-0007.cnf	SAT	180	0.047389	0	0.023536
examples/uf5-23/uf5-0008.cnf	UNSAT	210	0.030243	210	0.030429
examples/uf5-23/uf5-0009.cnf	SAT	165	0.038886	0	0.019152
examples/uf5-23/uf5-0010.cnf	UNSAT	210	0.031820	210	0.031509
examples/uf6-27/uf6-0001.cnf	SAT	571	1.020632	0	0.504794
examples/uf6-27/uf6-0002.cnf	UNSAT	664	0.732346	664	0.745252
examples/uf6-27/uf6-0003.cnf	SAT	512	0.752005	301	0.450002
examples/uf6-27/uf6-0004.cnf	SAT	571	1.039088	0	0.513455
examples/uf6-27/uf6-0005.cnf	SAT	602	1.177210	0	0.591881
examples/uf6-27/uf6-0006.cnf	SAT	509	0.724244	52	0.358600
examples/uf6-27/uf6-0007.cnf	SAT	547	0.895252	90	0.443119
examples/uf6-27/uf6-0008.cnf	SAT	602	1.186968	0	0.581589
examples/uf6-27/uf6-0009.cnf	UNSAT	664	0.739790	664	0.736167
examples/uf6-27/uf6-0010.cnf	SAT	602	1.158529	0	0.583362

Table 2 shows the number of clause sets and the runtime for resolution and DP on some small batches. DP eliminates variables and simplifies the problem and, in some cases, it even reduces

the formula to an empty set, correctly identifying satisfiability or unsatisfiability early. Resolution, on the other hand, tends to introduce a large number of intermediate clauses, especially when the problem requires deeper inferences. This results in exponential growth in clause count and, as shown in the runtimes, a dramatic performance drop for Resolution. Although both algorithms are complete, DP is clearly more efficient in practice for small instances.

For DPLL we observed very fast execution for the small batches, so we made a separate comparison for systems up to 200 variables. Resolution and DP were performing very poorly on systems with 9 variables, so we did not include them in further tests.

Table 3: Runtime comparison (seconds) for DPLL on larger SAT instances

Algorithm	**05 / 23**	**10 / 45**	**20 / 91**	**75 / 325**	**100 / 450**	**150 / 675**	**200 / 900**
DPLL	0.000028	0.000083	0.000356	0.150648	4.319880	351.480639	12735.369274

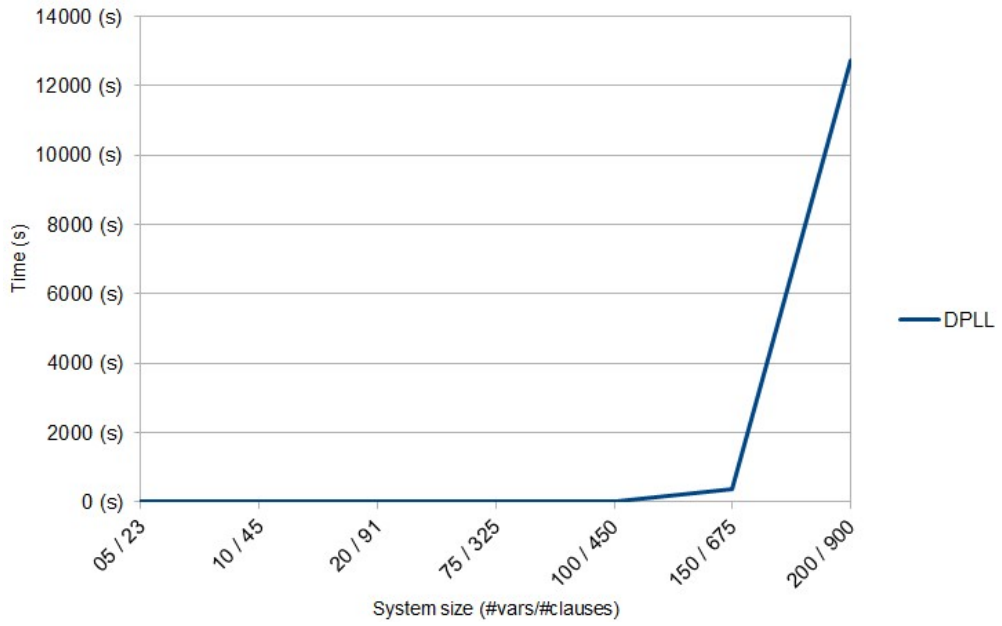

Figure 2: DPLL runtimes chart

The experimental results show notable differences in scalability:

- **Resolution** experiences exponential blow-up in both time and space due to excessive clause generation.
- **Davis-Putnam** improves over pure resolution but still struggles with memory constraints and variable ordering.
- **DPLL** exhibits the best performance, especially on larger or structured inputs, due to effective search space pruning.

4.3 Limitations of Classical Approaches

We are now going to highlight the main drawbacks for each method:

Resolution Method Limitations

- **Clause explosion**: The number of generated resolvents grows exponentially.
- **Lack of guidance**: The method doesn't focus on the unsatisfiability core.

Davis-Putnam Limitations

- **Memory usage**: New clauses that are generated can still overwhelm memory.
- **Variable ordering sensitivity**: Performance depends heavily on the chosen elimination sequence.

DPLL Limitations

- **Redundant exploration**: Without learning, similar conflicts are rediscovered multiple times.
- **Heuristics needed**: Basic versions lack intelligent variable selection.

5 Modern SAT Solving Techniques

In the previous section, we implemented and analyzed Resolution, DP and DPLL, without any improvements, to observe their strengths and limitation. We concluded that DPLL is by far the best method, but that there is still room for a lot of improvement. The algorithm lacks a way of selecting variables in a more efficient and intelligent manner. Modern solvers introduce several powerful techniques that drastically improve performance.

In this section, we are going to present the state-of-the-art and analyze the newest SAT solving techniques.

Modern SAT solvers are typically built around the CDCL (Conflict-Driven Clause Learning) paradigm [12]. This method improves DPLL with several optimizations:

- **Conflict Analysis and Clause Learning:** When a conflict is detected, the solver analyzes it and learns a new clause that prevents the same conflict from happening again. This drastically reduces redundant work.
- **VSIDS Heuristic:** Variable State Independent Decaying Sum is a dynamic heuristic that prioritizes recently involved variables in conflicts [13], so the search focuses on active areas of the formula.
- **Watched Literals:** An efficient technique to implement unit propagation by tracking only two literals per clause [13], to reduce the overhead of clause scanning.
- **Restart Policies:** Periodically restarting the search process helps escape hard regions of the search space and improves overall performance [10].
- **Clause Database Management:** Solvers actively manage the database of learned clauses, removing less useful ones to maintain performance [2].

Recent solver innovations also use **inprocessing techniques**, such as variable elimination, subsumption, and blocked clause elimination [3].

There is also increasing interest in **portfolio solvers** and **parallel solving**, where multiple strategies are run in parallel or selected based on the features of the input formula [9].

In fact, there is an annual conference (The International Conference on Theory and Applications of Satisfiability Testing [1]) where advances in the field are presented. They also organize a competition of SAT solvers on three categories:

- Main track (sequential algorithms)
- Parallel track (parallel algorithms)
- Cloud track (massive parallel algorithms in the cloud)

Top solvers from the SAT Competition 2024 [14], such as *Kissat*, *MapleSAT*, and *CaDiCaL*, also demonstrate highly tuned implementations of these ideas presented above and more. Many modern solvers are specialized for certain formula types (e.g., satisfiable, unsatisfiable, random, or industrial), using tailored heuristics or preprocessing strategies to excel in their categories.

5.1 Case Study: Kissat Solver

Kissat (Keep It Simple Stupid Advanced Tool) is a highly optimized CDCL-based SAT solver developed by Armin Biere. It consistently ranks among the top solvers in the SAT Competition series, including the 2024 edition [4]. Kissat is a clean, efficient, and industrial-strength solver written in C, designed with modern performance heuristics in mind.

Kissat is build with all the standard CDCL techniques. Its efficiency is the result of meticulous optimization of each core component. Some of the key features of Kissat are:

- **Aggressive Inprocessing:** inprocessing techniques like variable elimination, subsumption, and blocked clause elimination are applied dynamically throughout solving.
- **Efficient Restarts:** hybrid and adaptive restart strategies are used to regularly reset the search while preserving useful learned clauses.
- **Tuned Clause Management:** Learned clauses are cleaned with careful heuristics to balance memory use and relevance.
- **Literal Block Distance (LBD):** LBD is used to rank learned clauses by usefulness [2], with low LBD clauses retained longer.
- **Proof Logging and Verification:** Kissat supports proof generation (e.g., in DRAT format) to make it suitable for use in formally verified systems.

In SAT Competition 2024 [14], Kissat maintained top performance in multiple tracks, especially in the Main and Unsatisfiable tracks. Its development emphasizes both theoretical soundness and practical speed, making it one of the current state-of-the-art solvers for industrial benchmarks.

6 Conclusions and Future Work

This project explored the Boolean Satisfiability Problem (SAT) theoretically and practically. We began with a formal definition of SAT and an overview of its importance in complexity theory and real-world applications, then we implemented and evaluated three classical algorithms: Resolution, Davis–Putnam (DP) and Davis–Putnam–Logemann–Loveland (DPLL). Among these, DPLL was the most effective in terms of both clause reduction and runtime, especially on satisfiable formulas.

Resolution, while sound and complete, was significantly less efficient in practice because of its high clause overhead.

Despite our success with small to medium benchmarks, the classical solvers struggled with larger and more complex formulas. This motivated our analysis of modern techniques such as conflict-driven clause learning (CDCL) and heuristic variable selection, as used in state-of-the-art solvers like Kissat. These solvers vastly outperform classical methods by learning from conflicts, maintaining decision levels, and dynamically adjusting strategies.

However, we did not implement CDCL or other advanced optimizations ourselves. This remains open work.

In conclusion, while classical SAT solvers are useful for educational and analytical purposes, modern techniques are essential for solving realistic problems. Future work will focus on bridging this gap by incrementally integrating CDCL features into our framework and analyzing their impact on performance and solver robustness.

References

[1] *Proceedings of the 27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, Pune, India, August 21–24 2024. Available at `https://satisfiability.org/SAT24/`.

[2] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. *IJCAI*, 9:399–404, 2009.

[3] Armin Biere. Inprocessing vs. preprocessing in sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 389–393. Springer, 2009.

[4] Armin Biere, Kevin Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Thomas Pollitt. Sat competition 2024: Solvers and their contributions, 2024. Available at: `https://cca.informatik.uni-freiburg.de/papers/BiereFallerFazekasFleuryFroleyksPollitt-SAT-Competition-2024-solvers.pdf`.

[5] Stephen A. Cook. The complexity of theorem-proving procedures. *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC)*, pages 151–158, 1971.

[6] Adrian Crăciun. *Logic for Computer Science*. 2022. `https://staff.fmi.uvt.ro/~adrian.craciun/lectures/logic/pdf/logicNotes.pdf`.

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

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

[9] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2011.

[10] Jingchao Huang. Empirical study of restart strategies. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 216–220. Springer, 2007.

[11] Massimo Lauria, Marc Vinyals, Jan Elffers, Mladen Mikša, and Jakob Nordström. Cnfgen: A generator for cnf formulas. `https://massimolauria.net/cnfgen/`, 2017. Project by Massimo Lauria (`massimo.lauria@uniroma1.it`), with contributions by Marc Vinyals, Jan Elffers, Mladen Mikša, and Jakob Nordström.

[12] Joao Marques-Silva and Karem A Sakallah. Conflict-driven clause learning sat solvers. In *Handbook of satisfiability*, pages 131–153. IOS Press, 2009.

[13] Matthew W Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.

[14] SAT Competition Organizers. Sat competition 2024, 2024. Available at: `https://satcompetition.github.io/2024/`.