SAT solving: Davis-Putnam vs. Resolution

Mihai Vlad Marcu
Department of Computer Science,
West University of Timișoara
email: mihai.marcu05@e-uvt.ro

May 10, 2025

Abstract

This paper presents a comparative study of two classical SAT solving algorithms: Resolution and Davis-Putnam (DP). Both algorithms are implemented in Python and evaluated using identical inputs in Conjunctive Normal Form (CNF). Execution times are measured using Python's "time" library to assess performance.

The results demonstrate a clear advantage in efficiency for the Davis-Putnam algorithm over the Resolution method on the selected test inputs.

Contents

1 Introduction

Satisfiability (SAT) solvers have long been a subject of study in computer science, especially due to the problem having been proven to be NP-complete. SAT has profound implications for practical applications such as hardware verification and artificial intelligence.

Over the years, a large amount of algorithms have been developed to tackle SAT problems. Some of the most well-known are Resolution, Davis-Putnam (DP), Davis-Putnam-Logemann-Loveland (DPLL) and Conflict-Driven Clause Learning (CDCL). Each of these have unique strategies, bonuses and trade-offs in terms of efficiency, implementation complexity and time complexity.

This paper focuses on a comparative analysis of two foundational SAT solving methods: Resolution and the Davis-Putnam algorithms. Both algorithms operate on formulas expressed in Conjunctive Normal Form (CNF), a standard representation of a SAT problem. To compare both algorithms, we implement both methods in the programming language Python and evaluate their performance on a set of benchmark formulas, measuring the execution time as the primary metric. The Resolution and Davis-Putnam algorithms are well-established methods in the field of SAT solving. However, the Python implementation of both algorithms, as well as the test inputs used for the performance evaluation, were developed elusively by the author, Mihai Vlad Marcu, as original work for this research.

2 The Problem and the Solution

2.1 The SAT Problem

The **Boolean Satisfiability Problem (SAT)** is defined as follows:

Given a Boolean formula **F** composed of variables (such as letters of the alphabet), logical connectives (such as AND, OR, NOT), and parentheses, determine whether there exists an assignment of truth values (true or false) to the variables that makes the entire formula result in tautology.

Formally, SAT is the decision problem of determining the satisfiability of a formula $\mathbf{F} \in \mathbf{P}$, where $\mathbf{P}$ is the set of propositional logic formulas. SAT was the first problem proven to be **NP-complete**.

2.2 Conjunctive Normal Form (CNF)

In practice, SAT solving algorithms often require formulas to be in **Conjunctive Normal Form (CNF)**, which is a conjunction of clauses, where each clause is a disjunction of literals. A *literal* is a Boolean variable, represented by a letter of the alphabet, or its negation. For example:

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

This form allows solvers to apply rule-based methods more effectively.

2.3 The Resolution Algorithm

The **Resolution algorithm** is a proof-based method for determining the unsatisfiability of a CNF formula. It works by iteratively applying the *resolution rule*:

$$\frac{(x \vee A),\ (\neg A \vee y)}{x \vee y}$$

This rule states that if two clauses contain complementary literals (e.g., A and $\neg A$), a new clause can be created by combining the remaining literals. If the empty clause is eventually created, the formula is unsatisfiable.

If a formula is unsatisfiable, resolution will eventually derive a contradiction. However, it may be inefficient in practice due to the exponential growth in the number of derived clauses.

2.4 The Davis-Putnam Algorithm

The **Davis-Putnam (DP)** algorithm is a Resolution-based method that also operates on CNF formulas. It eliminates variables using the *Pure Literal* and *One Literal* rules, followed by resolution if none of the rules are applicable anymore.

Key operations include:

- **One Literal Rule**: If a clause has a single literal, assign it to satisfy the clause, remove any clause containing the literal. Remove all appearances of the complementary literal from all clauses.
- **Pure Literal Rule**: If a literal appears with only one polarity, it can be assigned to make all its clauses true, allowing us to safely remove them.
- **Resolution**: If neither of the two rules are applicable, apply Resolution. After every new clause created, check if the other two rules apply.

Unlike the basic Resolution method, DP simplifies the formula iteratively until it either finds a contradiction (UNSAT) or an assignment (SAT).

3 Resolution, DP Python Implementation

The Python implementation operates on input provided in Conjunctive Normal Form (CNF). Each formula is parsed as a list of clauses, where each clause is represented using square brackets and contains a comma-separated sequence of literals. A literal is defined as a single letter (representing a variable), optionally preceded by an exclamation mark to indicate negation (e.g., `A`, `!B`).

The program assumes that the literals within each clause are sorted alphabetically and that no clause contains a pair of complementary literals (e.g., both `A` and `!A`). These assumptions simplify parsing and reduce the risk of tautologies during resolution.

This list of clauses is stored in a global variable, as it is accessed by all major functions within the program. Additionally, a second global variable is used to indicate which algorithm is active: a value of `1` enables the Davis-Putnam (DP) algorithm, while a value of `0` enables the Resolution algorithm.

The input string is parsed by a dedicated function that scans the characters and splits them into individual clauses and literals, converting the input into a nested list structure suitable for logical manipulation.

To measure execution time, the program uses Python's `time` library. It records the time immediately before and after each algorithm's execution. A one-second delay is introduced before execution begins to improve readability in the results. The final output displays the total time taken by the selected algorithm, including the one-second delay.

3.1 Resolution Algorithm

The implementation of the Resolution algorithm consists of two main functions. The first function iterates through all pairs of clauses in the current clause list. For each distinct pair, it calls the second function to attempt resolution. If the result is either `NOT_COMPLEMENTARY` (indicating no complementary literals were found) or a tautology (i.e., the first clause is completely complementary to the second), the function skips it and continues to the next pair. Otherwise, the newly derived clause is added to the list of clauses if its not already present in the clause list.

The second function performs the actual resolution between two clauses. It first splits each clause into a list of individual literals. It then searches for a pair of complementary literals, one from each clause. If a pair is found, both literals are removed. The remaining literals from both clauses are then combined to form a new clause. If the resulting clause is empty, it indicates that a contradiction has been reached, and the original formula is unsatisfiable. In that case, the function returns `UNSATISFIABLE`. Otherwise, it returns the newly resolved clause which is sorted alphabetically.

3.2 Davis-Putnam Algorithm

The Davis-Putnam (DP) algorithm introduces two simplification rules in addition to the use of resolution: the *One Literal Rule* and the *Pure Literal Rule*.

The One Literal Rule applies when there exists a clause that contains only a single literal. In this case, the value of that literal can be set to true. All clauses containing that literal are removed, and its complementary literal is removed from any remaining clauses.

The Pure Literal Rule applies when a literal appears in the formula without its complement. In such cases, the literal can also be assigned true, and all clauses containing it can be eliminated. These rules are applied repeatedly as long as they remain applicable. If neither rule applies, the algorithm falls back to Resolution. After each clause addition during Resolution, the program checks if either simplification rule has become applicable again.

The Python implementation is composed of several functions, each responsible for a specific component of the algorithm:

- The first function serves as a controller, repeatedly applying the simplification rules in order. It checks after each application whether another rule can be used again.
- The second function implements the One Literal Rule. It scans the clause list for any clause containing a single literal. Once found, it stores the literal, removes all clauses containing it, and removes its complement from the remaining clauses. If any clause becomes empty in the process, the function immediately returns `UNSATISFIABLE`. If the list of clauses becomes empty after repeated applications, it returns `SATISFIABLE`.
- The third function handles the Pure Literal Rule. It constructs a set of all literals in the clause list, then removes any literal that has its complement also present. If no pure literals remain, the function returns `NO PURE LITERALS`. Otherwise, it removes all clauses containing the remaining pure literals. If this results in an empty clause list, the function returns `SATISFIABLE`; otherwise, it returns `PURE LITERAL APPLIED`.

The full source code, along with instructions for use, is publicly available at the project's GitHub repository [1].

4 Case Study and Experimental Analysis

To better understand the behavior of the two SAT solving algorithms, we analyze how each one handles the same input example. The following clauses are used:

`[A,B],[!A,C],[!A]`

This CNF formula consists of three clauses and serves as a compact, illustrative example.

4.1 Resolution Algorithm Walkthrough

The Resolution algorithm attempts to derive a contradiction by applying the resolution rule between pairs of clauses. In this case:

1. Resolve `[A,B]` and `[!A,C]`:

   $$\frac{(A \vee B),\ (\neg A \vee C)}{(B, C)}$$

   New clause: `[B,C]`, add to the clause list. The list now looks like: `[A,B],[!A,C],[!A],[B,C]`

2. Resolve `[A, B]` and `[!A]` on the literal `A`:

   $$\frac{(A \vee B),\ (\neg A)}{B}$$

   New clause: `[B]`, add to the clause list. The list now looks like: `[A,B],[!A,C],[!A],[B,C],[B]`

3. Resolve `[A,B]` and `[B,C]`: No complementary literal pair.
4. Resolve `[A,B]` and `[B]`: No complementary literal pair.
5. Resolve `[!A,C]` and `[!A]`: No complementary literal pair.
6. Resolve `[!A,C]` and `[B,C]`: No complementary literal pair.
7. Resolve `[!A C]` and `[B]`: No complementary literal pair.
8. Resolve `[!A]` and `[B,C]`: No complementary literal pair.
9. Resolve `[!A]` and `[B]`: No complementary literal pair.
10. Resolve `[B,C]` and `[B]`: No complementary literal pair.
11. No contradiction is found, and no empty clause is derived.

Therefore, the formula is `SATISFIABLE` according to the Resolution procedure.

4.2 Davis-Putnam Algorithm Walkthrough

The Davis-Putnam (DP) algorithm in this implementation follows an iterative process. It repeatedly applies two simplification rules—the One Literal Rule and the Pure Literal Rule—until neither is applicable. If no conclusion has been reached, the algorithm then applies Resolution to the remaining clause set. After each new clause is added through Resolution, the One Literal Rule is immediately retried.

Using the input:

```
[A,B],[!A,C],[!A]
```

the algorithm proceeds as follows:

1. **One Literal Rule:** The clause `[!A]` is a unit clause.
   - Save the literal `!A`.
   - Remove all clauses containing `!A`: `[!A,C]` and `[!A]` are removed.
   - Remove the complementary literal `A` from all remaining clauses: `[A,B]` becomes `[B]`.
2. The rule is applied again: `[B]` is now a unit clause.

- Save `B`.
- Remove all clauses containing `B`: `[B]` is removed.
- There are no more clauses left to update.

3. **Pure Literal Rule:** Not needed—clause list is already empty.
4. The clause set is now empty, so the formula is `SATISFIABLE`.

Throughout execution, the program continuously applies each rule as long as it is applicable. If neither rule yields progress, Resolution is invoked. After each new clause added via Resolution, the One Literal Rule is reattempted. This iterative design allows the solver to efficiently simplify the formula before resorting to resolution steps.

5 Conclusion and Future Work

5.1 Results and Conclusion

The results of this study demonstrate a consistent performance advantage of the Davis-Putnam (DP) algorithm over the Resolution algorithm when applied to a variety of SAT problems. Execution times were measured using Python's `time` library, and each test case was executed 10 times to account for runtime variability. The final results for each algorithm and input were computed as the average of these 10 runs to ensure accuracy.

Initially, a one-second delay was introduced before each measurement to improve the human readability of the output, as very small runtime values would otherwise be displayed in exponential notation. However, for the purpose of precise benchmarking, this artificial delay was subtracted from the final timing results reported in Table 1.

Formula	**Resolution**	**DP**
[!A,!W,P],[A,I],[W,M],[!P],[!S,!I],[!S,!M] - SAT	0.0004594	0.0003666
[A,!B,C],[B,C],[!A,C],[B,!C],[D,E],[!D,B,C] - UNSAT	0.001013	0.0004209
[A,B],[!A,C],[!B,!C] - SAT	0.0005235	0.0001695
[A,B],[!A,C],[!B,!C],[!A,!C] - SAT	0.0006505	0.0003217
[P,Q,!R],[!P,R],[P,!Q,S],[!P,!Q,!R],[P,!S] - SAT	0.2269385	0.0076376
[A,B],[!A,C],[!A] - SAT	0.0001069	0.0000783
[!A,!B,!C],[D,E,!A],[D,B],[C] - SAT	0.0005054	0.0001516

Table 1: Average execution times (in seconds) for Resolution and Davis-Putnam (DP) algorithms

As shown in Table 1, the Davis-Putnam algorithm is faster than Resolution in all tested cases. For more complex formulas, such as those involving multiple variables, the gap between the two algorithms becomes more pronounced. In one notable case, DP completed execution in roughly 0.007 seconds, whereas Resolution took over 0.22 seconds,a difference of more than an order of magnitude.

These results align with the theoretical expectations: by applying simplification rules like the One Literal Rule and the Pure Literal Rule before falling back on Resolution, the DP algorithm effectively reduces the complexity of the problem. This simplification reduces the number of clauses that need to be resolved, leading to faster solving times.

Overall, the experiment confirms the advantage of Davis-Putnam in practical execution speed for small-to-medium SAT problems, validating its relevance as a more efficient alternative to pure Resolution in many use cases.

5.2 Future Work

While this paper focused on comparing the Resolution and Davis-Putnam (DP) algorithms in terms of execution time on CNF inputs, there remains room for further exploration. One promising direction is to extend the study to include the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, a refinement of DP that introduces backtracking and variable selection. DPLL is widely regarded as the foundation for modern SAT solvers and offers significant performance improvements in many cases.

A future paper could implement and benchmark DPLL alongside DP, by applying them to a diverse set of test formulas. This comparison would provide deeper insight into how backtracking mechanisms influence execution time, especially on larger or more complex SAT problems.

References

[1] Mihai Vlad Marcu. *DP and Resolution SAT Implementation.* GitHub repository. `https://github.com/Marcu-Mihai-Vlad/DP-program-py`, 2025.