SAT: What teachers don't tell you about solving efficiently SAT problems

Delinschii Vladislav
vladislav.delinschii05@e-uvt.ro
West University of Timisoara

April 2025

1 Introduction

The Boolean Satisfiability Problem (SAT) asks whether a given Boolean formula can evaluate to TRUE by assigning TRUE/FALSE values to its variables. A formula is *satisfiable* if such an assignment exists, and *unsatisfiable* otherwise. As the first problem proven NP-complete, SAT is central to theoretical computer science and complexity theory. Its importance extends to artificial intelligence and other domains, where problems like planning or verification can be encoded as SAT instances. The inherent computational difficulty of this NP-complete problem makes developing effective solving algorithms crucial.

This paper focuses on comparing two foundational algorithms developed relatively early in SAT research: the Davis-Putnam (DP) procedure and its influential refinement, the Davis-Putnam-Logemann-Loveland (DPLL) algorithm. DP typically utilized variable elimination via resolution, while DPLL introduced the backtracking search paradigm fundamental to many modern solvers, crucially relying on branching heuristics.

Our goal is to present both a theoretical overview and an experimental comparison of DP and DPLL. We implemented these algorithms in TypeScript, paying particular attention to different choice strategies within DPLL. The experiments aim to analyze their performance characteristics on benchmark instances. The following sections detail the theoretical background, implementation, experimental setup, results, and conclusions.

2 Background and Preliminaries

This section outlines the fundamental concepts and algorithms relevant to Boolean Satisfiability (SAT) solving, forming the basis for the comparisons presented in this paper.

2.1 The Boolean Satisfiability Problem (SAT)

The Boolean Satisfiability Problem (SAT) asks whether a given Boolean formula F, composed of Boolean variables and logical connectives (AND, OR, NOT), can evaluate to TRUE by assigning consistent TRUE/FALSE values to its variables. If such an assignment exists, the formula F is deemed *satisfiable*; otherwise, it is *unsatisfiable*. For example, the simple formula $(a \land \neg b)$ is satisfiable by assigning $a =$ TRUE and $b =$ FALSE.

SAT holds significant importance as the first problem proven to be NP-complete. Its NP-completeness signifies both its theoretical centrality and its practical difficulty, as no known algorithm solves all SAT instances efficiently in the worst case. Despite this hardness, SAT serves as a powerful modeling tool, enabling the encoding of numerous problems from artificial intelligence (e.g., planning, constraint satisfaction) and other computer science domains like hardware verification.

2.2 Conjunctive Normal Form (CNF)

Most modern SAT solvers operate on formulas expressed in Conjunctive Normal Form (CNF). A formula is in CNF if it is structured as a conjunction (AND) of one or more *clauses*, where each clause is a disjunction (OR) of one or more *literals*. A literal is simply a Boolean variable (e.g., x_i) or its negation (e.g., $\neg x_i$). For instance, the formula $(x_1 \lor \neg x_2) \land (\neg x_1 \lor x_3)$ is in CNF, consisting of two clauses joined by AND. Representing problems in CNF is standard practice, as any Boolean formula M can be converted into an equisatisfiable CNF formula.

2.3 Algorithms for SAT Solving

Over the years, various algorithms have been developed to tackle SAT. We focus on three foundational approaches:

2.3.1 Resolution

Resolution is a sound and complete inference rule primarily used for proving *unsatisfiability*. The rule states that given two clauses containing complementary literals, a new clause (the resolvent) can be inferred by combining the literals from both parent clauses, excluding the complementary pair. Formally:

$$\frac{(A \lor x) \quad (B \lor \neg x)}{(A \lor B)}$$

where A and B represent disjunctions of other literals. A resolution-based procedure repeatedly applies this rule. If the empty clause (containing no literals, denoted , representing a contradiction) can be derived, the original formula is declared unsatisfiable. While complete, resolution can suffer from generating an exponential number of intermediate clauses.

2.3.2 Davis-Putnam (DP) Algorithm

The original Davis-Putnam (DP) algorithm (often dated to 1960) primarily tackled SAT using **variable elimination** through resolution. It iteratively selected a variable x and performed all possible resolutions between clauses containing x and clauses containing $\neg x$. After generating these resolvents, all original clauses containing x or $\neg x$ were discarded, effectively eliminating x. While effective for some problems, this approach could lead to significant memory consumption due to the potentially large number of generated resolvents.

2.3.3 Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm (often dated to 1962), a significant refinement, replaced the costly variable elimination step of DP with a recursive backtracking search. It forms the basis of many successful modern SAT solvers. The core DPLL procedure involves applying simplification rules followed by a splitting rule:

1. **Unit Propagation (or BCP - Boolean Constraint Propagation):** If a clause contains only one unassigned literal (a unit clause), that literal must be assigned the value required to satisfy the clause (TRUE if positive, FALSE if negative). This assignment typically triggers further simplifications (satisfying other clauses, shortening clauses containing the complementary literal), and the process iterates until no more unit clauses exist.

2. **Pure Literal Elimination:** If a variable appears only with one polarity (e.g., x appears but $\neg x$ does not) across all *currently active* clauses, that variable can be assigned the value TRUE (FALSE if only $\neg x$ appears). Clauses satisfied by this assignment are then removed. (This rule is sometimes omitted in modern implementations).

3. **Splitting Rule (Branching):** If neither unit propagation nor pure literal elimination simplifies the formula further, DPLL selects an unassigned variable x. It then recursively attempts to solve the formula first by assuming x = TRUE. If this leads to unsatisfiability, it *backtracks* and recursively attempts to solve the formula assuming x = FALSE. This branching step explores the space of possible assignments.

The recursion terminates when either all clauses are satisfied (result: SAT) or an empty clause is generated via simplification (result: UNSAT).

2.4 The Role of Choice Heuristics in DPLL

The efficiency of the DPLL algorithm's splitting rule heavily depends on the strategy used for choosing the next variable to branch on. Because the search space can be exponentially large, a good heuristic can significantly prune the search tree by leading to contradictions or solutions more quickly. Various heuristics exist, from simple ones like selecting the first unassigned variable to more complex strategies like choosing the variable involved in the most unresolved clauses. The comparison of such strategies is a key focus of this paper's experimental section.