A Theoretical and Experimental Comparison of DP and DPLL Algorithms for the Boolean Satisfiability Problem

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

April 2024

Abstract

The Boolean Satisfiability (SAT) problem, a canonical NP-complete problem, is central to computer science and artificial intelligence. This paper explores methods for solving SAT, focusing on a theoretical and experimental comparison between the classic Davis-Putnam (DP) algorithm and Davis-Putnam-Logemann-Loveland (DPLL) algorithm. I implemented both algorithms in TypeScript, incorporating unit propagation, pure literal elimination, and multiple branching heuristics (First Available, Random, MOMS) within DPLL. Experiments were conducted on standard CNF benchmarks to evaluate performance based on execution time and, for DPLL, the number of branching decisions. Results confirm DPLL's significant practical advantages over the basic DP approach, which suffered from memory limitations, due to its resolution-based variable elimination. With DPLL, the MOMS heuristic demonstrated a better performance in reducing the search space compared to simpler heuristics, highlighting the critical impact of choice strategies in tackling SAT. The implementation and experimental data are publicly available.

1 Introduction

The Boolean Satisfiability Problem (SAT) asks whether there exists an assignment of TRUE/FALSE values to the variables of a given Boolean formula such that the formula evaluates to TRUE. If such an assignment exists, the formula is deemed *satisfiable*; otherwise, it is *unsatisfiable* [6]. As the first problem proven NP-complete [7], SAT holds a position of fundamental importance in theoretical computer science and complexity theory. Its NP-completeness implies that, assuming P $\neq$ NP, no algorithm can solve all SAT instances efficiently in the worst case.

Beyond its theoretical significance, SAT solving has become indispensable in numerous practical domains. Its power lies in the ability to encode a vast array of discrete constraint satisfaction problems into the simple, uniform language of Boolean logic. Consequently, advances in SAT solving algorithms directly translate into improved capabilities for tackling complex challenges in areas such as hardware and software verification, artificial

intelligence (AI), planning, scheduling, cryptography, and bioinformatics [6]. The continued growth in these application areas constantly drives the need for more efficient and robust SAT solvers.

This paper revisits two foundational algorithms that went the way for modern SAT solving: the Davis-Putnam (DP) procedure and its influential, refined algorithm, the Davis-Putnam-Logemann-Loveland (DPLL). DP, originating around 1960, primarily utilized variable elimination via the resolution inference rule. while complete, this approach often suffered from large memory requirements. DPLL, developed shortly after in 1962, introduced the recursive backtracking search, incorporating crucial simplification techniques like **unit propagation**, which remains central to contemporary solvers. A key aspect of DPLL's performance, and a focus of this work, is the impact of *heuristics* used to guide the branching choices during the search.

Our goal is to provide both a theoretical overview and an experimental comparison of DP and DPLL. We have implemented these algorithms in TypeScript (§3), integrating several distinct choice strategies within the DPLL framework (§3.3). Through experiments conducted on standard benchmark instances (§4), we aim to quantitatively analyze and compare the performance characteristics of these approaches (§5, §6). The paper concludes (§7) by summarizing the findings and reflecting on the trajectory of SAT solving research.

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)

Formally, the Boolean Satisfiability Problem (SAT) takes a Boolean formula F as input. This formula is constructed using a set of Boolean variables $V = \{x_1, x_2, ..., x_n\}$ and logical connectives (typically AND ($\wedge$), OR ($\vee$), and NOT ($\neg$)). The question is whether there exists a truth assignment $\tau : V \to \{\text{TRUE}, \text{FALSE}\}$ such that F evaluates to TRUE under τ. If such a τ exists, F is *satisfiable*; otherwise, F is *unsatisfiable* [6]. For example, the formula $F = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$ is satisfiable (e.g., $\tau(x_1) = \text{TRUE}, \tau(x_2) = \text{TRUE}$), while $G = (x_1 \vee x_2) \wedge (\neg x_1) \wedge (\neg x_2)$ is unsatisfiable.

The significance of SAT cannot be overstated. Its status as the seminal NP-complete problem establishes it as a benchmark for computational hardness. Any problem in the class NP can be polynomially reduced to SAT, meaning an efficient SAT solver could, in principle, efficiently solve any problem in NP [7]. This theoretical centrality is mirrored by its practical relevance. The ubiquity of SAT stems from its expressive power; numerous computational problems across diverse fields can be naturally and often efficiently encoded as SAT instances. Notable examples include:

- **Planning and Scheduling:** Finding valid sequences of actions or resource allocations often involves satisfying complex logical constraints [6].

- **Hardware Verification:** Checking if a digital circuit design meets its specification or contains errors can be modeled by checking the satisfiability of a formula representing the circuit's behavior under certain conditions.
- **Software Verification:** Bounded model checking and other techniques use SAT solvers to find bugs or prove properties of software systems.
- **Constraint Satisfaction Problems (CSPs):** Many finite-domain CSPs can be translated into equivalent SAT problems.
- **Cryptography:** Analyzing the security of certain cryptographic primitives can sometimes involve solving related SAT instances.

Solving the derived SAT instance effectively provides a solution to the original problem, making SAT solvers powerful general-purpose reasoning engines [6].

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 $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3)$ is in CNF, consisting of two clauses joined by AND. Representing problems in CNF is standard practice, as any Boolean formula can be converted into an equisatisfiable CNF formula, often using methods like the De Morgan. Solvers typically consume CNF formulas represented in the standard DIMACS format.

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:

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

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 (§3.3, §5, §6).

3 Implementation Details

This section details the implementation choices made for the DP and DPLL algorithms in TypeScript, executed using Node.js.

3.1 Core Data Structures

The core data structures used are defined in `types.ts`:

- `Literal`: Represented as a standard JavaScript `number`. Positive numbers denote positive literals (e.g., 1 for x_1), and negative numbers denote negative literals (e.g., -1 for $\neg x_1$).
- `Clause`: Implemented as a `Set<Literal>`. The choice of `Set` offers two main advantages: efficient literal existence checks (average O(1) complexity for `has()`), very good for simplification rules, and automatic handling of duplicate literals within a clause(you can't have duplicates in a Set).
- `CnfFormula`: Represented as an array of `Clause` objects (`Clause[]`).
- `Assignment`: A partial assignment is stored in a `Map<number, boolean>`, mapping a variable number (always positive) to its Boolean truth value. Unassigned variables are simply absent from the map.

3.2 DPLL Algorithm

The DPLL solver is implemented in the recursive function `dpll`. It follows the standard structure:

1. Base cases check for an empty formula (SAT) or a formula containing an empty clause (UNSAT).
2. **Unit Propagation:** The `applyUnitPropagation` function is called. It iteratively finds unit clauses, updates the assignment, and simplifies the formula using `simplifyFormula`. If a conflict arises (empty clause generated during simplification), it returns a 'CONFLICT' state.
3. **Pure Literal Elimination:** The `applyPureLiteralRule` function identifies and assigns pure literals, further simplifying the formula via `simplifyFormula`.
4. **Branching:** If no conflict occurred and the formula is not empty, a variable selection heuristic (§3.3) chooses the next variable `v`. The `dpll` function is called recursively, first assuming `v = TRUE` (after simplifying the formula accordingly). If this branch returns UNSAT, it backtracks and calls itself assuming `v = FALSE`.

A counter (`decisionCounter`) passed through the recursion tracks the number of branching steps.

3.3 DPLL Heuristics

Three variable selection heuristics were implemented for the DPLL branching step:

- `selectVariable_FirstAvailable`: A simple baseline selecting the lowest-numbered unassigned variable.
- `selectVariable_Random`: Selects an unassigned variable uniformly at random.
- `selectVariable_MOMS`: Implements the Maximum Occurrences in clauses of Minimum Size heuristic. It finds the shortest active clause length and selects the unassigned variable appearing most frequently in clauses of that minimum length. This aims to prioritize highly constrained variables.

3.4 DP Algorithm

The classic Davis-Putnam algorithm is implemented in the `dpSolver` function. It operates iteratively:

1. **Simplification Loop:** Repeatedly applies DP-specific unit propagation (`applyDPUnitPropagati`
and pure literal elimination (`applyDPPureLiteralElimination`) until no further simplification occurs in a full pass (a fixed point is reached). Checks for immediate SAT (empty formula) or UNSAT (empty clause generated) are performed after each step.
2. **Variable Selection:** If the formula is not yet solved, the smallest-numbered variable present in the formula is chosen for elimination.
3. **Resolution Step:** All possible resolvents are generated between clauses containing the positive literal of the chosen variable and clauses containing the negative literal. Tautological resolvents are discarded.
4. **Formula Update:** The new formula consists of the clauses not containing the eliminated variable, plus the newly generated, non-tautological resolvents. Duplicate clauses are removed.
5. **Termination:** The process repeats until the formula is empty (SAT) or an empty clause is derived (UNSAT). Stalemate conditions (where resolution produces no new information) and limits on clause count (`MAX_CLAUSES_DP = 500000`) and iterations are included to handle potential non-termination or memory explosion, reporting specific statuses (`DP_CLAUSE_LIMIT`, `DP_MAX_ITERATIONS`).

As expected from the theory, this DP implementation does not efficiently reconstruct a satisfying assignment and can consume significant memory, potentially failing on larger instances due to clause set explosion.

3.5 Benchmarking Harness

The `runBenchmarkHarness` orchestrates the experiments. It reads benchmark filenames from `data/benchmarks.list`, loops through each file and each algorithm/heuristic combination (DP, DPLL-first, DPLL-random, DPLL-MOMS), calls the respective solver via `runSolver`, and collects results (`SolverResult` interface). `runSolver` measures execution time using Node.js's `performance.now()` and manages the DPLL decision counter. Results are printed to the console and saved to `benchmark_results.csv`.

3.6 Code Availability

The complete TypeScript implementation is available on GitHub: `https://github.com/biscuitdelicious/MPI_SAT.git`

4 Experimental Setup

To evaluate and compare the implemented DP and DPLL algorithms (with its heuristics), we conducted experiments using a set of standard benchmark instances.

4.1 Benchmark Suite

The benchmark suite was configured via the `data/benchmarks.list` file and included instances primarily from the SATLib repository [5]. We selected a mix of satisfiable (uf) and unsatisfiable (uuf) instances from the uf20/uf100/uuf250 series, representing problems with 20 variables/91 clauses, 100 variables/430 clauses, etc.

Small custom instances were also included for basic correctness checks.

4.2 Environment

Experiments were performed on a system with the following specifications:

- CPU: M1 Pro
- RAM: 16GB
- OS: macOS Sequoia
- Node.js Version: v20.18.0

4.3 Procedure and Metrics

Each CNF file listed in `data/benchmarks.list` was processed by:

1. The DP solver (`dpSolver`).

2. The DPLL solver (`dpll`) using the 'first' heuristic.
3. The DPLL solver using the 'random' heuristic.
4. The DPLL solver using the 'MOMS'(Maximum Occurrences in clauses of Minimum Size) heuristic.

For each run, the following metrics were recorded by the `runBenchmarkHarness`:

- **Execution Time:** Wall-clock time in milliseconds (ms), measured using `performance.now()`.
- **Decision Count:** The number of times the splitting rule was invoked in the DPLL algorithm (N/A for DP, reported as 0).
- **Final Status:** The outcome reported by the solver: `SATISFIABLE`, `UNSATISFIABLE`, `DP_CLAUSE_LIMIT`, `DP_MAX_ITERATIONS`, or error statuses like `PARSING_FAILED`.

The results were aggregated and saved to `benchmark_results.csv`.

5 Results

This section presents the key results obtained from running the DP and DPLL solvers on the benchmark suite described in §4. Detailed results are available in the accompanying CSV file.

Table 1: Comparison of Solver Performance on Selected Benchmarks

Benchmark File	Solver/Heuristic	Status	Time (ms)	Decisions
dp_sat_2.cnf	DP	SATISFIABLE	1.40	0
	DPLL-first	SATISFIABLE	0.33	1
	DPLL-random	SATISFIABLE	0.18	1
	DPLL-moms	SATISFIABLE	0.13	1
dp_unsat_3.cnf	DP	UNSATISFIABLE	0.69	0
	DPLL-first	UNSATISFIABLE	0.30	0
	DPLL-random	UNSATISFIABLE	0.06	0
	DPLL-moms	UNSATISFIABLE	0.04	0
uf20-0995.cnf	DP	DP_LIMIT (exp.)	N/A	N/A
	DPLL-first	SATISFIABLE	1.64	24
	DPLL-random	SATISFIABLE	0.64	10
	DPLL-moms	SATISFIABLE	0.90	8
uf100-0999.cnf	DP	DP_LIMIT (exp.)	N/A	N/A
	DPLL-first	SATISFIABLE	4044.08	16105
	DPLL-random	SATISFIABLE	4600.77	19051
	DPLL-moms	SATISFIABLE	57.61	161
uuf125-083.cnf	DP	DP_LIMIT (exp.)	N/A	N/A
	DPLL-first	UNSATISFIABLE	44717.76	129175
	DPLL-random	UNSATISFIABLE	157858.23	462808
	DPLL-moms	UNSATISFIABLE	334.93	694

Key observations from the results include:

- The DP solver consistently failed to complete on instances larger than 20 variables, hitting the clause or iteration limit, indicating its memory/scalability issues.
- The DPLL solver successfully solved all tested instances within the uf20/uuf50 sets.
- Comparing DPLL heuristics, MOMS generally resulted in the lowest number of decisions.
- In terms of execution time for DPLL, MOMS was often the fastest, despite its higher overhead per decision, followed by first and then random. The random heuristic showed that sometimes it's much faster than choosing the first, while it doesn't guarantee always the success..

6 Discussion

The experimental results presented in §5 highlight several important aspects of the compared SAT solving algorithms and the nature of the SAT problem itself.

The NP-completeness of SAT implies that no known algorithm guarantees efficient solutions for all possible instances in the worst case. Finding a satisfying assignment, if one exists, can require exploring a potentially exponential search space. However, the remarkable practical success of solvers like DPLL and its modern descendants (such as Conflict-Driven Clause Learning - CDCL) on large, industrially relevant instances demonstrates that worst-case complexity does not always translate to poor average-case performance, especially on problems with inherent structure [6]. This practical utility continues to drive research into both algorithmic improvements and novel application domains.

Our experiments vividly illustrate the performance gap between the classic DP algorithm and the DPLL approach. The DP solver's reliance on the resolution rule, while theoretically complete, leads to a practical bottleneck.

DPLL, using a backtracking search approach, avoids storing a potentially massive set of intermediate clauses.

Limitations of this study include the use of a basic DP implementation lacking sophisticated optimizations like subsumption...

7 Conclusion

This paper presented a theoretical overview and an experimental comparison of the Davis-Putnam (DP) and Davis-Putnam-Logemann-Loveland (DPLL) algorithms for solving the Boolean Satisfiability problem, a cornerstone challenge in computer science and AI. We implemented both algorithms in TypeScript, including three distinct branching heuristics (First Available, Random, MOMS) for DPLL, to investigate the impact of algorithmic structure and choice strategies.

My experiments, conducted on standard CNF benchmarks, confirmed the practical limitations of the basic DP algorithm due to memory constraints arising from its resolution-based variable elimination, aligning with historical observations. In contrast, the DPLL algorithm successfully solved the tested instances, demonstrating the advantage of its backtracking search framework. The comparison of DPLL heuristics underscored the significant impact of the choice strategy, with the MOMS heuristic generally achieving the best performance by effectively reducing the search space (measured by decision count), which often translated to faster execution times compared to simpler strategies.

These findings reinforce the importance of the DPLL algorithm's structure and the value of informed branching heuristics in tackling the NP-complete SAT problem. While DPLL forms the basis for many solvers, modern systems typically incorporate further enhancements like Conflict-Driven Clause Learning (CDCL), advanced heuristics (e.g., VSIDS), and random restarts. As AI and automated reasoning continue to advance, the need for robust and efficient SAT solvers remains in its place, ensuring its continued relevance as a core area of research and a powerful tool for diverse applications [6]. Future work could involve extending the current implementation to include CDCL features or exploring a wider range of modern heuristics.

References

References

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

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

[3] Lecture 4: DPLL. COMP6463, Australian National University. Accessed [Date Accessed]. `https://users.cecs.anu.edu.au/~tiu/teaching/comp6463/lecture4.pdf`

[4] Boolean Satisfiability Solvers: Techniques and Implementations. Course Notes (e.g., 15-820A), Carnegie Mellon University. Accessed [Date Accessed]. `https://www.cs.cmu.edu/~emc/15-820A/reading/sat_cmu.pdf`

[5] Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. In *SAT 2000*, pages 283-292. IOS Press, 2000. `http://www.satlib.org`

[6] Autoblocks AI. Boolean Satisfiability Problem. Glossary Entry. Accessed [May 2 2025]. `https://www.autoblocks.ai/glossary/boolean-satisfiability-problem`

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