Comparative Analysis of Boolean Satisfiability Algorithms

Fabfive

This project presents a comprehensive comparative analysis of five Boolean Satisfiability (SAT) solvers: Baseline DPLL, DPLL with Jeroslow-Wang (JW-TS) heuristic, Conflict-Driven Clause Learning (CDCL), WalkSAT, and probSAT. All solvers were implemented entirely **from scratch** in Python without reliance on external SAT libraries. We benchmarked these algorithms against a diverse suite of Random 3-SAT instances (50–200 variables) and structured Sudoku encodings. Our empirical results demonstrate that CDCL outperforms DPLL by orders of magnitude on structured problems (0.16s vs 2.83s for Sudoku) due to effective search space pruning via clause learning. Conversely, Stochastic Local Search (SLS) solvers proved superior for large satisfiable random instances, solving 200-variable problems in $\approx$4.5s (10% success rate) where complete solvers consistently timed out. On small random instances (50 variables), DPLL variants achieved 100% success, while CDCL struggled with initialization overhead.

1 Introduction

1.1 Problem Definition

The Boolean Satisfiability Problem (SAT) is the problem of determining if there exists an interpretation (variable assignment) that satisfies a given Boolean formula. Formally, given a formula ϕ in Conjunctive Normal Form (CNF), SAT asks whether there exists an assignment $A : \{x_1, ..., x_n\} \to \{0, 1\}$ such that $\phi(A) = 1$. SAT is the archetypal problem for the class **NP-complete**.

1.2 Project Objectives

The primary objectives of this study were to:

1. Implement complete (DPLL, CDCL) and incomplete (WalkSAT, probSAT) solvers from scratch.

2. Analyze the performance of these solvers across two distinct problem domains: Random 3-SAT at the phase transition ($m/n \approx 4.26$) and Structured (Sudoku) instances.
3. Evaluate the impact of modern heuristics, including the Jeroslow-Wang heuristic, VSIDS variable ordering, and Clause Learning (1-UIP).

2 Algorithm Descriptions & Theoretical Analysis

2.1 Baseline DPLL

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is a recursive backtracking search. Our implementation (`dpll_baseline.py`) includes two critical optimizations:

- **Unit Propagation**: If a clause contains only one unassigned literal, it must be set to true immediately to reduce the search space.
- **Pure Literal Elimination**: Literals that appear with only one polarity in the formula are assigned that polarity.

Complexity: Worst-case time complexity is $O(2^n)$, with space complexity $O(n)$ due to recursion depth.

2.2 DPLL with Jeroslow-Wang (JW-TS)

Implemented in `dpll_jw.py`, this variant improves decision making using the Two-Sided Jeroslow-Wang heuristic. It prioritizes literals that appear in short clauses to maximize the likelihood of a conflict or unit propagation. The score $J(l)$ for a literal l is calculated as:

$$J(l) = \sum_{C \in \phi, l \in C} 2^{-|C|} \tag{1}$$

This heuristic significantly reduces the search tree depth compared to random or static variable selection by targeting the most constrained variables first.

2.3 Conflict-Driven Clause Learning (CDCL)

Our `cdcl.py` implements the modern standard for complete SAT solvers. Unlike DPLL, CDCL does not backtrack chronologically but learns from conflicts. Key ”from-scratch” components include:

- **2-Watched Literals**: A lazy data structure that reduces the overhead of visiting clauses during propagation, significantly improving speed per decision.

- **1-UIP Learning**: Upon conflict, the implication graph is analyzed to find the First Unique Implication Point (1-UIP), generating a learned clause that prevents the same conflict pattern from reoccurring.
- **VSIDS Heuristic**: Variable State Independent Decaying Sum maintains a score for every variable, bumped during conflict analysis and decayed over time ($decay = 0.95$), focusing the search on "active" constraints.

2.4 Stochastic Local Search (SLS)

2.4.1 WalkSAT

`walksat.py` implements a randomized incomplete solver. It starts with a random assignment and iteratively flips variables. With probability p (noise), it flips a random variable in an unsatisfied clause (escaping local minima); otherwise, it performs a "greedy" flip to minimize the number of broken clauses.

2.4.2 probSAT

`probsat.py` uses a fully probabilistic selection mechanism. It calculates a "break score" for each variable in an unsatisfied clause and selects a variable to flip based on a polynomial probability distribution function. This approach avoids the hard distinction between greedy and random moves found in WalkSAT, often leading to more robust convergence on specific problem types.

3 Implementation Details

3.1 Design Choices

- **Language**: Python 3.12 was selected for readability and rapid prototyping.
- **Architecture**: The project uses a modular design with a dedicated `harness/` for reproducibility and `solvers/` for isolated algorithmic logic.
- **Data Structures**:
  - **VSIDS**: Implemented using Python's `heapq` module for $O(\log n)$ priority queue updates.
  - **Watches**: Adjacency lists were used for Watched Literals to minimize cache misses.
  - **State**: Python `dataclasses` were used to track solver state (decision levels, assignment trail) cleanly.

3.2 Benchmark Generation

The project includes a `utils` module for instance generation:

- **Random 3-SAT**: Generated using `generate_benchmarks.py`. We specifically targeted the phase transition ratio of clauses-to-variables $m/n \approx 4.26$. This ratio is mathematically proven to produce the computationally hardest instances for SAT solvers.
- **Sudoku Encoder**: `sudoku_encoder.py` reduces 9x9 Sudoku puzzles to CNF. The encoding enforces four constraints: (1) One number per cell, (2) Unique numbers per row, (3) Unique numbers per column, and (4) Unique numbers per 3x3 box. This results in exactly 729 variables and $\approx$12,000 clauses per puzzle.

4 Results & Analysis

4.1 Experimental Setup

Experiments were conducted using the `run_experiments.py` harness. The setup included:

- **Timing**: Both Wall-clock time (`perf_counter`) and CPU time (`process_time`) were captured. A strict timeout of 60 seconds was enforced.
- **Memory Profiling**: We used the `tracemalloc` library to capture Peak Memory usage, allowing us to compare the memory footprint of recursive DPLL versus the iterative, structure-heavy CDCL.
- **Correctness Verification**: The harness automatically verified every satisfying assignment returned by a solver against the original CNF clauses to ensure validity.

4.2 Scalability: Random 3-SAT

We evaluated performance on random 3-SAT instances generated at the phase transition ($m/n = 4.26$) with sizes ranging from 50 to 200 variables.

- **Small Instances (50 vars)**: DPLL variants achieved 100% success ($\approx 0.10s$). CDCL struggled (25% success) due to the overhead of initializing watched literals and learning structures on trivial problems.
- **The Wall (150+ vars)**: A clear scalability limit was observed. Complete solvers (DPLL, CDCL) failed to scale, consistently timing out on 200-variable instances.

- **SLS Dominance**: WalkSAT and probSAT were the only solvers capable of handling 200 variables within the timeout, achieving a 10% success rate in $\approx 4.5s$. This confirms that incomplete solvers are essential for large, unstructured instances.

4.3 Robustness: Sudoku

Our experiments on structured Sudoku instances revealed a dramatic performance inversion compared to random SAT.

- **CDCL Superiority**: CDCL solved Sudoku instances in an average of **0.16s** with 100% success.
- **DPLL Lag**: Baseline DPLL averaged **2.83s**. The 17x speedup of CDCL confirms that clause learning effectively exploits the hidden structure (backdoor sets) in Sudoku puzzles.
- **SLS Failure**: Stochastic solvers (WalkSAT, probSAT) failed completely (0% success). Without the ability to propagate unit constraints, SLS algorithms cannot navigate the "narrow canyons" of highly constrained logical puzzles like Sudoku.

4.4 Heuristic Efficiency

Internal metric tracking revealed the impact of heuristics on search space reduction:

- **Decision Counts**: CDCL required only ≈4.5 decisions on average for solvable structured problems, compared to ≈2,000 for Baseline DPLL.
- **Implication**: This reduction of $> 99\%$ proves that the VSIDS heuristic combined with non-chronological backtracking allows the solver to skip vast, irrelevant sections of the search tree.

4.5 Parameter Sensitivity

We analyzed the impact of the noise parameter p in WalkSAT. Sweeping values of $p \in \{0.1, 0.3, 0.5, 0.7\}$ revealed an optimal setting of $p = 0.5$. Lower values ($p = 0.1$) caused the solver to get stuck in local minima, while higher values ($p = 0.7$) degraded the search into a random walk, increasing convergence time.

5 Conclusion

This comparative analysis highlights the "No Free Lunch" theorem in the context of SAT solving:

1. **CDCL** is the superior choice for **structured problems** (like Sudoku), where learning from conflicts allows it to prune vast sections of the search space.
2. **SLS Solvers** (WalkSAT, probSAT) are essential for **large, satisfiable random instances**, providing solutions where complete solvers time out.
3. **Heuristics** are fundamental; the performance gap between Baseline DPLL and CDCL/SLS demonstrates that naive search is infeasible for non-trivial problem sizes.

Bonus Disclosures

The following features were implemented to exceed baseline requirements:

- **Full CDCL Implementation**: Includes 1-UIP learning, Non-chronological backtracking, and VSIDS heuristic.
- **Advanced SLS**: Implementation of `probsat.py` for comparison against WalkSAT.
- **Infrastructure**: Custom Sudoku-to-CNF encoder and automated Test Harness with memory profiling.
- **Analysis**: Rigorous parameter sensitivity sweep for WalkSAT noise.