# A Study on the Reproducibility of Boolean Satisfiability Algorithms (Team Teletubbies)

VLADIMIR MAKSIMOVSKI, SHREIF ABDALLAH, LIAM HEEGER, and CHRIS KJELLQVIST, University of Rochester

Reproducibility is a problem that has plagued modern research because future research may be based off research that is not necessarily truth. Effort to reproduce and understand important result is important to generate enough confidence in pivotal findings that they can be used in the future. Those findings, especially, that are considered groundbreaking need to be vetted.

We present a reproducibility study for various boolean satisfiability algorithms. We provide naïve backtracking as a baseline against the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, conflict-driven clause learning (CDCL), and Schöning's 3-SAT algorithm. The results show us that the performance for the algorithms matches the published data, but that under certain conditions, some algorithms may perform differently than others. We also discuss the tradeoffs associated with a probabilistic algorithm like Schönings.

## 1 INTRODUCTION

Scientific research has long been plagued with un-reproducible results and sometimes pure fabrication. Famously, several papers completely generated by computers were accepted to a computer science conference in 2005 [Stribling et al.]. These papers were accepted only as non-reviewed papers. Perhaps a more groundbreaking and important case was in the case of Daryl Bem. In his 2011 study he showed that extra-sensory perception (ESP) was a real phenomenon under the right circumstances [2] [4]. Wagenmakers et. al. showed in 2012 that showed that Bem was able to achieve these results through a practice called optional stopping [12]. Optional stopping is the practice of performing an experiment and continuing to collect results until the results show the desired result. Wagenmaker showed that this practice can result in completely arbitrary results and as a result can not be allowed in experiments. This work resulted in a new standard for experimentation and a new fervor to re-investigate developments in the field of human cognition research. In the field of computer science every couple of years, a paper surfaces offering a solution to the P versus NP problem [Woeginger] [1]. These proofs are sometimes easy, but sometimes extremely difficult to disprove, but have always been proven incorrect. The continuous progress of useful research relies on the ability of the community to not only peer review research, a relatively unintrusive process, but to occasionally spend significant amounts of resources in formally disproving results in the case of Bem and sometimes in the case of P vs NP proofs. For mathematical proofs, languages such as Coq have been introduced to allow for machine checked proofs

While the practice of peer review has become standard for a conference to be considered trustworthy, they are unable to detect the errors in Bem's results. For Bem, it required the development of a new theory on experimentation. While this of course is a good result, it shows that replication of results by peers is necessary to confirm the validity of important results. And for the plethora of P versus NP proofs, each one requires careful consideration before formally discarding.

Boolean satisfiability is one of the most fundamental NP-Complete problems in computer science. It asks if any arbitrary formulation of boolean variables and boolean operators has a satisfying assignment to the variables. Because of its NP-Completeness, this algorithm is relevant for many different fields and purposes, from pragmatic uses like protein folding, to mathematical problems like N-Queens. Therefore, it is absolutely pivotal that any important results be

Authors' address: Vladimir Maksimovski; Shreif Abdallah; Liam Heeger; Chris Kjellqvist, University of Rochester, Rochester, New York.

properly vetted so that research in other areas may continue unobstructed by the truthfulness of research in unfamiliar fields.

To establish some notation, we formally describe the SAT problem as follows: given are a set of boolean variables, and a set of clauses. Each clause contains a set of literals. A literal is a pair of a variable and a polarity (true or false). An assignment is a set of literals such that every variable has exactly one polarity. A clause is satisfied w.r.t an assignment if there is a literal in the clause which is also in the assignment. The set of clauses is satisfied w.r.t an assignment if and only if each clause is satisfied. The negation of a literal is a literal with the same variable but opposite polarity. A clause is unsatisfied w.r.t an assignment if the negation of every literal in the clause is in the assignment. The SAT problem asks whether an assignment exists which satisfies a given set of variables and set of clauses.

In this paper, we re-implement a variety of SAT algorithms and compare their efficiency. We have chosen DPLL [3], CDCL[8], and Shönings[7] as our targets because they represent a showing of both old and new algorithms and they serve as the backbone for modern techniques. We also implement a naïve backtracking algorithm to show the relative speedups of the algorithms against a baseline. We show that these algorithms have the expected performance as reported by their original papers and give an analysis on the tradeoffs for probabilistic algorithms such as Schönings.

## 2 SOLUTION TECHNIQUE

SAT algorithms have undergone much study over the past forty years. Since there are many algorithms implementing SAT, to reduce the scope of this project, we decided to focus only on the major solution techniques.

As a starting point for other algorithms, we implemented simple backtracking search. DPLL is somewhat of an optimization over backtracking, which has been a building block of many other SAT algorithms.

One such algorithm which used DPLL as a building block was the conflict-driven clause learner (CDCL) algorithm. This is the basis of the MiniSAT solver, an open-source SAT solver which has been the basis for many top-performing SAT algorithms [10].

### 2.1 Backtracking

Backtracking is a SAT solving technique which relies on recursion to arrive at a satisfying result for a Boolean formula. This method works by extending partial solutions to a Boolean formula to arrive at a satisfying assignment. When a partial solution is found to make the formula false, the algorithm "backtracks" to the previous partial assignment, trying a new value for a variable. This process continues until a satisfying assignment is found or all assignments are exhausted.

Backtracking is a poor SAT algorithm whose worst case runtime is $O(2^n)$, where n is the number of variables. That said, it is the foundation for other SAT algorithms such as DPLL and CDCL. These algorithms are the basis for state-of-the-art solvers. We will discuss their function in subsequent chapters.

### 2.2 Davis–Putnam–Logemann–Loveland (DPLL)

As previously stated, DPLL uses a similar approach to the naïve backtracking algorithm expect that it optimizes the set of clauses before each iteration. The two optimizations that DPLL makes are the propagation of pure literals and unit clauses. As a note, we assume that variables that are assigned and clauses that are already satisfied are removed from the set of clauses.

A literal is pure if the negation of the literal does not exist within the set of clauses. For instance, $a$, and not $b$ are pure in the following CNF clauses: $(a \lor b) \land (a \lor \neg b)$. The value in pure literals is that by making that pure literal part of the assignment, some clauses will now be satisfied and all other clauses will not become harder because they necessarily never contained the negation of that literal.

Unit clauses are clauses which only contain a single literal. Therefore in order for the clause to be satisfied, that literal must be part of the assignment.

The algorithm begins by first accepting if the input is empty and rejecting as unsatisfiable if the input contains any empty clauses. As we noted, satisfied clauses and assigned variables are removed, and therefore an empty clause represents a clause that can not be satisfied by any assignment to any variable. Propagation of unit clauses and pure literals is then performed. If at any point, a clause becomes empty, the entire set of clauses is unsatisfiable. After this, simply choose an unassigned variable and assign a value to it in the same way as the backtracking algorithm.

While clearly this algorithm will perform better than backtracking in practice because it will never iterate over variables that are trivially implied by assignments to previous variables, the worst case remains the same: $O(2^n)$ . This is easily realized because the optimizations in the worst case do nothing, in which case this is just the backtracking algorithm.

## 2.3 Conflict-driven clause learning (CDCL)

Conflict-driven clause learning expands on the unit propagation done in DPLL. It does not remove satisfied clauses or assigned variables from clauses.

The unit propagation used in CDCL works as follows: If a clause has n literals, and n - 1 of the variables in the clause have been assigned, and the clause remains unsatisfied, then the remaining unassigned literal is forced to hold. We say that the remaining unassigned literal has been *propagated.* This is essentially an implication, with the antecedents being the negations of the n - 1 pre-assigned literals in the clause, and the consequent being the remaining literal.

CDCL follows a similar structure to the backtracking algorithm. At each step, an unassigned variable is chosen according to some heuristic, a literal (X = true or X = false) is added to the assignment, and CDCL proceeds recursively. After a literal is chosen, CDCL checks if any clause can be unit-propagated.

If a clause can be unit-propagated, we add the propagated literal to the assignment. Adding this literal might reveal more unit-propagations, so we search for more unit-propagations. We store the set of all antecedents and consequents in a set $S$.

After the clauses are propagated, it's possible that some clause becomes unsatisfiable. This is called a conflict. We consider the set of antecedents and consequents $S$. If another propagation performs the same propagations, this will result in the same conflict occuring again. CDCL avoids repeating this conflict by adding $S$ as a new clause in the clause list.

Once the conflict is detected, CDCL backtracks.

Due to the learned clause, CDCL doesn't repeat the same conflict twice. So the earlier a conflict occurs in the backtracking process, the more work is saved. This enlightens how we should choose our heuristic: we should choose a heuristic that finds conflicts as early as possible!

The current heuristic for choosing a new variable to assign is to pick the variable involved in the maximal number of unsatisfied clauses. If the incorrect polarity for a variable is assigned, then it will take few steps to reveal the conflict, resulting in many skipped subtrees! Instead, if the correct polarity for the variable is assigned, then we have just satisfied many clauses!

This shows the strength of the CDCL approach, and why it's one of the top-performing algorithms.

### 2.4 Schöning’s K-SAT algorithm

The key difference between Schöning’s K-SAT algorithm and the previously explored algorithms (Backtracking, DPLL, CDCL) is that it is randomized and non-deterministic. The algorithm maintains the same goal, of finding a satisfying assignment of truth values to variables within a disjunction of clauses, with some limitations. The first limitation is that the algorithm is non-deterministic, meaning that it may not find a solution if one exists, and cannot prove the absence of a solution if the set of clauses is unsatisfiable. The second limitation is that if it finds a certain assignment of truth variables to values within clauses such that all clauses are satisfied, it may find a different admissible solutions on repeated runs of the program.

The algorithm starts out by assigning a random truth value {true, false} to each unassigned variable in the n-variable set, and then finds a clause that is unsatisfied, and flips the truth value of one of its literals randomly, repeating the process 3n times, where n is the number of variables in the problem. If a satisfying assignment occurs at any point of the program, the program stops. [7] Therefore, the program can either return an assignment satisfying all clauses or it fails in finding one. If the program fails in finding a satisfying assignment, this is no conclusion that no such assignment can be found, as there might be a satisfying assignment that the program could have found if it had better luck on flipping the appropriate variables’ truth values.

The original paper by Uwe Schöning concludes that the complexity of k-SAT is within a polynomial factor of $O(2(1-\frac{1}{k}))^n$ [7], for the 3-SAT problem, this polynomial factor becomes $O(\frac{4}{3})^n$ as compared to $2^n$ by naive backtracking.

## 3 EVALUATION

### 3.1 Correctness

To ensure correctness of each algorithm, we sourced multiple SAT examples, and compared each algorithm’s output with the correct output. We also generated many thousands of random examples, and compared each algorithm’s output.

All deterministic algorithms (backtracking, DPLL and CDCL) were determined to be correct. When Schöning’s algorithm returned a result, it was determined to always be equal to that of other 3 algorithms. Of course, as it is a probabilistic algorithm, it sometimes returned "I don’t know" for certain inputs, which was not considered incorrect.

The user can evaluate correctness by running the `SATSolver/test_correctness.sh` script, which will run the SATSolver algorithms, and check for incorrect results.

### 3.2 Performance

To benchmark performance, we used two input data types: a random set of inputs which the program generates, and the uf benchmark. We’ll describe each of these in turn. We benchmarked performance on the `node2x18a.csug.rochester.edu` machine, for better reproducibility.

*3.2.1 Random inputs.* To evaluate these algorithms, we generated many random SAT examples, varying the number of clauses and variables between 1 and 120. The parameters of the experiment shown below are: 4-SAT instances, with number of clauses and number of literals varying between 1 and 120.

Test case generation was very simple: for a K-sat problem, each clause got K randomly chosen literals. This possibly meant a clause would contain $X = true$ and $X = false$, which indicates a tautology, and the clause will always be satisfied. Since none of the algorithms have hard-coded behavior in this case, we still believe all four algorithms are on even ground, and so we can perform relative comparisons.

To generate the benchmark data and diagrams, run `SATSolver/test_random.sh`. Results are shown in the chart below.

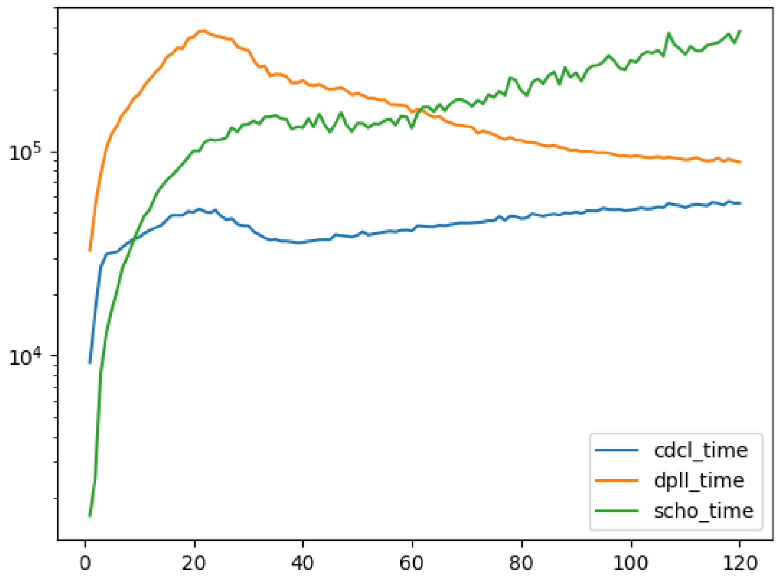

Fig. 1. Random 4-SAT instances with varying clause and variable sizes. X-axis shows number of variables in clause, while Y-axis shows the execution time in nanoseconds.

For small variable counts, DPLL is the slowest, followed by Schönings, and finally CDCL is the quickest. Meanwhile, for larger variable counts, the order changes to Schönings being the slowest, then DPLL, and again CDCL being the quickest. Notice that even though Schönings changes from being the quickest to the slowest, it's running time is very much a function of the number of trials performed, so it's difficult to reason about the argument's performance just using this diagram. However, when comparing the two deterministic algorithms, CDCL is quicker in most cases.

*3.2.2 Uniform Random-3-SAT.* We also used external inputs to benchmark our algorithms. For this, we used the SATLIB - Benchmark Problems dataset [11]. These are suites of 100-1000 SAT instances for variable sizes between 20 and 250. Due to time limitations, we ran the faster 3 algorithms for sizes between 20-175.
To generate the benchmark data and diagrams, run `SATSolver/test_uf.sh`. Note that this script takes more than 10 hours to finish, so it's recommended that the user modifies the script file's "benchmarks" vector to only contain uf20-91 and uf50-218.

Results are shown in the chart below.

The charts show that DPLL is slowest, then CDCL, and finally Schönings is the quickest. This makes sense, as CDCL is an optimization of DPLL, while Schönings is a probabilistic algorithm that sometimes outputs "I don't know". Manual

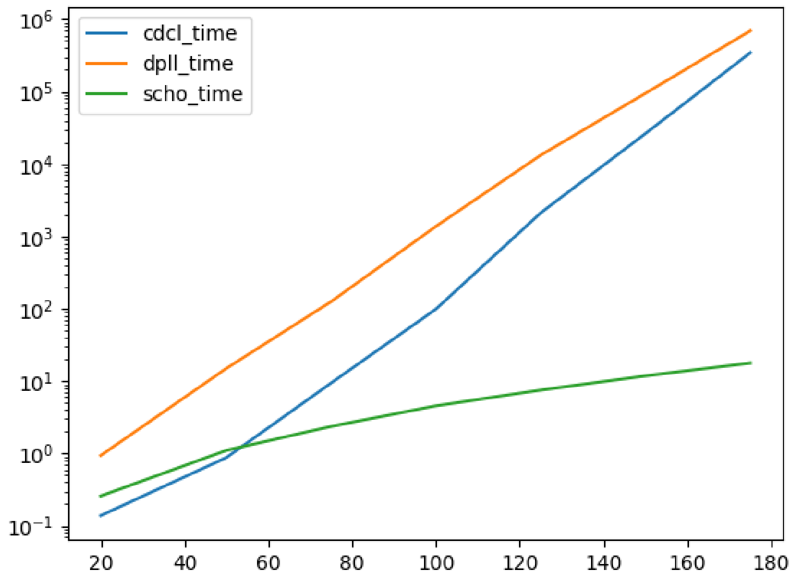

Fig. 2. SATLIB instances with varying variable sizes. X-axis shows number of variables in clause, while Y-axis shows the execution time in milliseconds.

inspection of `SATSolver/uf_run.csv` shows that Schönings very commonly outputs "I don't know" for this dataset. The takeaway from this experiment is that there is a significant difference between non-deterministic and deterministic SAT algorithms on large variable sizes. If a definite answer is needed, then CDCL algorithm is recommended.

## 4 RELATED WORK

In deciding how to approach a reproducability study for SAT solvers, we assessed other similar studies. These studies look at experimental analysis of the runtime of SAT algorithms. These studies use different experimental approaches, both on random problem distributions and practical subsets of problems.

The first paper we introduce is by Mladen Nikolić, the paper addresses the challenging task of comparing two arbitrary SAT solvers, and proposes a statistically founded methodology for comparing the relative performance of SAT solving algorithms. [6] This study is important because it outlines the shortcomings of comparing the relative performance of SAT solvers based solely on their runtime and number of solved instances, and proposes an alternative route that reduces chance effect and highlight the positive or negative effect of a modification.

Current literature also includes surveys with larger scope, displaying an array of novel ideas to the classic satisfiblity problem. [5], the survey also outlines a hierarchy of SAT solvers, including categories such as complete algorithms, that can decide weather there is a solution to the given problem, and incomplete algorithms such as Schöning's, that may not find a solution if one exists.

## 5 SUMMARY

We presented an analysis of the performance of four popular algorithms to solve the Boolean satisfiability problem. Our starting point was backtracking, which relied on recursion to reach a satisfying assignment of truth values to Boolean variables, this algorithm proved to be a poor solution with a worst case runtime of $O(2^n)$.

We then explored different modifications, improvements and optimizations on backtracking, arriving at the DPLL algorithm. The propagation of pure literals and unit clauses which is done by DPLL demonstrated an improved performance in practice. However, the theoretical worst case runtime remained $O(2^n)$, because in the worst case these optimizations might not change the outcome, therefore the DPLL algorithm becomes equivalent to backtracking.

To improve on the results of DPLL, we reimplemeneted conflict driven clause learning (CDCL), which is an optimization on DPLL where we don't remove satisfied clauses or assigned variables from clauses. This proved to provide the best performance of all backtracking-based algorithms, albeit reatining the same worst case time complexity.

We moved on to a different approach, which is a non-deterministic, probabilistic approach with a faster runtime. We arrived at Schöning's K-SAT algorithm, which relies on an initial random assignment which is iteratively improved by random changes of unsatisfied clauses. This approach had a runtime of $O(\frac{4}{3})^n$. While this is faster than the backtracking-based algorithms, it is not always guaranteed to find a solution if one exists, or to rule out the existence of a solution if one does not exist.

## 6 ACKNOWLEDGEMENTS

We thank professor Chen Ding and the TAs for their insightful advice on this project.

## REFERENCES

[1] (2017). A solution of the P versus NP problem. *CoRR*, abs/1708.03486. Withdrawn.

[2] Bem, D. J. (2011). Feeling the future: experimental evidence for anomalous retroactive influences on cognition and affect. *Journal of personality and social psychology*, 100(3):407.

[3] Davis, M., Logemann, G., and Loveland, D. (1961). *A machine program for theorem-proving.* Courant Institute of Mathematical Sciences, New York University.

[4] Engber, D. (2017). Daryl bem proved esp is real.

[5] Gong, W. and Zhou, X. (2017). A survey of sat solver. *AIP Conference Proceedings*, 1836(1):020059.

[6] Nikolić, M. (2010). Statistical methodology for comparison of sat solvers. In Strichman, O. and Szeider, S., editors, *Theory and Applications of Satisfiability Testing – SAT 2010*, pages 209–222, Berlin, Heidelberg. Springer Berlin Heidelberg.

[7] Schoning, T. (1999). A probabilistic algorithm for k-sat and constraint satisfaction problems. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, pages 410–414.

[8] Silva, J. P. M. and Sakallah, K. A. (1996). Grasp—a new search algorithm for satisfiability. In *International Conference on Computer-Aided Design (ICCAD)*, pages 220–227.

[Stribling et al.] Stribling, J., Krohn, M., and Aguayo, D. Scigen - an automatic cs paper generator.

[10] Sörensson, N. and Een, N. (2005). Minisat v1.13-a sat solver with conflict-clause minimization. *International Conference on Theory and Applications of Satisfiability Testing.*

[11] Taylor, W., Cheeseman, P., and Kanefsky, B. (1995). Where really hard problems are. *Proceedings of the International Joint Conference of Artificial Intelligence*, 1.

[12] Wagenmakers, E.-J., Wetzels, R., Borsboom, D., van der Maas, H. L. J., and Kievit, R. A. (2012). An agenda for purely confirmatory research. *Perspectives on Psychological Science*, 7(6):632–638. PMID: 26168122.

[Woeginger] Woeginger, G. J. The p-versus-np page.