Review: On SAT modulo theories and optimization problems [6]

CS550 Formal Verification, Background Paper Report

Matthieu Bovel (matthieu.bovel@epfl.ch) Solal Pirelli (solal.pirelli@epfl.ch)

1 Introduction

Solvers for *SAT Modulo Theories (SMT)* can solve are widely used to solve *satisfiability* problems over many different theories. The reviewed paper introduces a variant of the *DPLL Modulo Theories (DPLL(T))* algorithm that enables not only to find a valid model M for a given formula F withing a given theory T, but also to make it *optimal* with respect to a cost function *f(M)*. This opens the gate to solving problems that were previously out of the scope of SMT solvers, such as Max-SAT and Max-SMT (for which a specific example is given with the radio bandwidth allocation problem).

In this review, we first give a quick introduction to the *DPLL* and *DPLL(T)* algorithms, before diving into the contribution that the reviewed paper makes: *DPLL(T) with strengthening*. We illustrate these with the formal definitions, with toy implementations in Scala available on GitHub [8] and by discussing the examples presented in the paper. Finally, we discuss the "Benchmarks" part and conclude with general thoughts on the paper.

2 Preliminaries

DPLL is an algorithm solving the *boolean satisfiability problem (SAT)*: finding an assignment of variables for a formula F such that it evaluates to true. It is formalized as a set of rules deriving a final *state* S from a start state S_0. This allows to reason easily about it and to prove properties such as termination.

A state either the final $FailState$ state indicating that no model was found, or a pair $F \mid\mid M$, where F is the formula to satisfy and M is a partial assignment of the variables of F to boolean values. F is given in *conjunctive normal form (CNF)* $C_1 \wedge \cdots \wedge C_n$ where each *clause* is a disjunction of *literals* $l_1 \vee \cdots \vee l_n$.

The applicable steps are:

- "Decide": assign a random value to a literal l and mark it as a *decision literal*, meaning that the value choice was arbitrary and could be changed later.
- "UnitPropagate": if there exists a clause $l_1 \vee \cdots \vee l_n$ such that all l_i are false in M expect one that is unassigned, then it assign it to true, as this is the only possibility to make the clause true.
- "Backtrack": if we have an invalid assignment, forget all assignments made since the last decision literal l^d and set its value to $\neg l$. If we cannot, then we have failed ("Fail").

"PureLiteral" is an additional rule that sets the values of all literals from F whose negations are not in F to true. In practice, it can be applied only once at the start of the algorithm, as it does not depend on the assignment.

Definition 1 (Classic DPLL) *As described in [7, p.941]:*

UnitPropagate:

$$M \mid\mid F, C \vee l \implies M\, l \mid\mid F, C \vee l \quad \text{if} \begin{cases} M \models \neg C \\ l \text{ is undefined in } M. \end{cases}$$

PureLiteral:

$$M \mid\mid F \vee l \implies M\, l \mid\mid F \vee l \quad \text{if} \begin{cases} l \text{ occurs in some clause of } F \\ \neg l \text{ occurs in no clause of } F \\ l \text{ is undefined in } M. \end{cases}$$

Decide:

$$M \mid\mid F \implies M\, l^d \mid\mid F \quad \text{if} \begin{cases} l \text{ or } \neg l \text{ occurs in a clause of } F \\ l \text{ is undefined in } M. \end{cases}$$

Fail:

$$M \mid\mid F, C \implies FailState \quad \text{if} \begin{cases} M \models \neg C \\ M \text{ contains no decision literals.} \end{cases}$$

Backtrack:

$$M\, l^d\, N \mid\mid F, C \implies M\, \neg l \mid\mid F, C \quad \text{if} \begin{cases} M\, l^d\, N \models \neg C \\ N \text{ contains no decision literals.} \end{cases}$$

To demonstrate the algorithm, we made a small toy implementation in Scala. It is purposely close the formal definition, which makes it inefficient but easy to read.

Example 1 (Classic DPLL) *Example derivations logged by our toy implementation:*

With $F = (x \vee y) \wedge (\neg y \vee z) \wedge (\neg x \vee \neg z)$:

$$\begin{aligned} \emptyset &\implies \text{(Decide)} \\ x^d &\implies \text{(UnitPropagate)} \\ z\ x^d &\implies \text{(UnitPropagate)} \\ \neg y\ \neg z\ x^d &\implies \text{(SUCCESS)} \end{aligned}$$

With $F = (\neg x \vee y) \wedge (\neg y \vee z)$

$$\begin{aligned} \emptyset &\implies \text{(PureLitterals)} \\ \neg x\ z &\implies \text{(Decide)} \\ y^d \neg\ x\ z &\implies \text{(SUCCESS)} \end{aligned}$$

With $F = (\neg x \vee y) \wedge (\neg x \vee \neg y) \wedge (\neg x \vee y)$

$$\begin{aligned} \emptyset &\implies \text{(Decide) :} \\ x^d &\implies \text{(UnitPropagate)} \\ \neg y\ x^d &\implies \text{(Backtrack)} \\ \neg x &\implies \text{(UnitPropagate)} \\ y\ \neg x &\implies \text{(SUCCESS)} \end{aligned}$$

The "Backtrack" rule can be replaced by the "Backjump" rule which changes the value of the literal that caused the conflict, instead of simply choosing last decision literal in chronological order as "Backtrack". *Conflict-driven* back-jumping enables to jump further and to undo several decisions at once, which is more efficient [7, p. 944].

Definition 2 (Basic DPLL) *Replacing the rule "Backtrack" in "Classic DPLL":*

Backjump:

$$M\, l^d\, N \mid\mid F, C \implies M\, l' \mid\mid F, C \quad \text{if} \begin{cases} M\, l^d\, N \models \neg C \\ \text{there exists a clause } C' \vee l' \text{ such that:} \\ \quad F, C \models C' \vee l' \text{ and } M \models \neg C', \\ \quad l' \text{ is undefined in } M \text{ and} \\ \quad l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } M\, l^d\, N. \end{cases}$$

In the previous algorithms, literals are only boolean variables. However, the same algorithm can be adapted to handle literals that are predicates described in other theories. For example, a literal could be $x = y$ in the theory of equality, or $x < 10$ in the theory of

linear arithmetic. The SAT solver does not know these theories, but it can interact with a $Solver_T$ to check that the current assignment M is consistent with respect to T.

This is done by adding 4 steps to "Basic DPLL"

- "Theory Learn": adds new clauses to the formula F as needed to *teach* the DPLL about some implications. For example, if DPLL sets the truth values of the $x = y$ literal to false and $y = x$ to true, this step for the theory of equality could add a clause $\neg(x = y) \vee (y = x)$ to inform the algorithm of this implication.
- "Theory forget": as the added clauses can always be re-deduced from F, they can also be forgotten if needed (used to save resources).
- "Theory Propagate": can set the truth values of undecided literals when their implied by the assigned values of existing literals in M.
- "Restart": enables the algorithm to forget the current assignment and start over (used for efficiency).

The theory T is also added to the state of the algorithm.

Definition 3 (DPLL Modulo Theories) *In addition to the rules of "Basic DPLL", without "PureLiteral" (and adapting "BackJump"):*

Restart:

$$T \mid\mid M \mid\mid F \qquad \Longrightarrow T \mid\mid \emptyset \mid\mid F$$

Theory Learn:

$$T \mid\mid M \mid\mid F \qquad \Longrightarrow T \mid\mid M \mid\mid F, C \quad \text{if} \begin{cases} \text{each atom of } C \text{ is in } F \text{ or in } M \\ F \models_T C. \end{cases}$$

Theory forget:

$$T \mid\mid M \mid\mid F, C \qquad \Longrightarrow T \mid\mid M \mid\mid F \quad \text{if} \begin{cases} F \models_T C. \end{cases}$$

Theory Propagate:

$$T \mid\mid M \mid\mid F \qquad \Longrightarrow T \mid\mid M\, l' \mid\mid F \quad \text{if} \begin{cases} M \models_T l \\ l' \text{ or } \neg l' \text{ occurs in } F \\ l \text{ is undefined in } M. \end{cases}$$

3 Body

The main contribution of the paper is the addition of a new rule to "DPLL Modulo Theories" allowing to gradually refine the theory T to a stronger theory:

Definition 4 (Strengthening DPLL Modulo Theories)

Theory Strenghten:

$$T \mid\mid M \mid\mid F \qquad \Longrightarrow T \wedge T' \mid\mid M \mid\mid F$$

The new step is probably best illustrated with the "Max-SAT" problem.

3.1 Application: Max-SAT

In the Max-SAT problem we are given a set of clauses where each clause is assigned a weight. The goal is to find a model that maximize the weight of the clauses that are true. To encode it, we add a literal p_i to each clause that can be set to true to "ignore" the clause, and we define a theory that impose a maximum bound on the weights of p_is set to true. To demonstrate this, we made simplified toy implementation of Strengthening DPLL Modulo Theories and tested it with the Max-SAT theory.

3.2 Application: Max-SMT and difference logic

The paper shows how the approach can be extended to Max-SMT using the example of weighted Max-SMT modulo Integer Difference Logic ("QF_IDL" in SMT terms), which consists of Boolean combinations of $a - b \leq k$ inequations with a, b integer variables and k an integer constant.

Integer Difference Logic is typically used for verification, but the paper shows it can also be useful in an unexpected context: radio frequency assignment [2]. The goal is to assign frequencies to radio links in a way that minimizes interference: some links must be at specific frequency distances, while others have "soft" constraints specifying a minimum distance that should ideally be respected.

Radio frequency assignment is encodable in Integer Difference Logic thanks to the authors' observation that the set of available frequencies for any given radio link can be seen as the disjoint union of four sets, each constraining the frequency k to be $n + mk \mid a \leq k \leq b$ with n, m, a, b integer constants. This observation is unfortunately not justified in the paper and not mentioned in the original paper describing the problem [2] either; it may be an artifact of practical considerations given that this problem is a real-world one.

The key idea is to encode these disjoint sets using on the one hand Boolean variables determining which set is picked and on the other hand integer variables defining the frequency modulo m. Constraints on the distance between frequencies of specific links can be naturally defined in Integer Difference Logic since they match the logic's inequality shape.

4 Benchmarks

The authors use two kinds of benchmarks: existing benchmark suites for Max-SAT and the radio frequency assignment example for Max-SMT.

4.1 Choice of benchmarks

There are three existing benchmark suites: "DIMACS", "Weighted DIMACS", and "Quasi-groups". While the paper does not provide sources, the first appears to be a subset of the benchmarks available on https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html. The second seems to be a variation of the first with weights randomly varying from 1 to 1000 instead of being set at 1. The final one was given to the authors in private correspondence by Felip Manyà.

The radio frequency assignment example consists of the 5 "sub-CELAR" examples from Table 3 of [2].

4.2 Choice of baselines

As baselines, the authors compare against Toolbar [5], Pueblo [9], and MiniSat+ [4]. These represent three different kinds of solvers: Toolbar solves weighted constraint satisfaction problems (and includes specializations for Max-SAT), Pueblo solves pseudo-Boolean problems, and MiniSat+ translates pseudo-Boolean problems into SAT. The authors note that these tools are all designed for a broader class of problems than their own DPLL(T).

The authors use two different configurations for their DPLL(T) solver, one general and one specialized with heuristics for Max-SAT. They use three configurations for Toolbar: the default one, one with a static branching heuristic, and one with a Jeroslow-Wang heuristic recommended by the Toolbar authors. For unknown reasons, they only use the general DPLL(T) configuration for the radio frequency problem, and only use the default Toolbar configuration for the benchmarks.

4.3 Results

Overall, DPLL(T) is the fastest solver among all tested for all Max-SAT benchmarks except three: Toolbar is 2x faster for the DIMACS "jnh" benchmark, MiniSat+ is 2x faster for the DIMACS "dubois" benchmark, and Pueblo is 9x faster for the DIMACS "hole" benchmark. General DPLL(T) handily wins in other benchmarks, especially in the privately obtained "Quasi-groups" benchmarks in which it beats Pueblo by 2x, MiniSat+ by 30x or so, and Toolbar because the latter times out after 10 minutes on almost all queries.

Somewhat surprisingly, the addition of specialized heuristics for DPLL(T) does not help on the benchmarks; aside from one query in which general DPLL(T) times out after 10 minutes but specialized DPLL(T) finds an assignment, specialized DPLL(T) loses on other benchmarks, by up to 5x.

In the radio frequency example, the "Jeroslow" version of Toolbar is 1.5-5x faster for the first four queries but DPLL(T) is 25% faster for the last, and biggest, one. DPLL(T) however handily beats the default and static versions of Toolbar.

The authors attempted a translation of the radio frequency queries into weighted Max-SAT and pseudo-Boolean problems to use Pueblo and MiniSat+, but these solvers timed out after a day (!) on the second smallest query, which Toolbar and DPLL(T) solve in under 2 minutes.

4.4 Implications

Overall, the results do not show that DPLL(T) is always more efficient, but they do show that Max-SMT can be used to solve well-known problems that are not typically thought of as SMT problems, such as the radio frequency problem, without the need for a specialized solver.

5 Conclusion

This paper proposes a variant of SMT in which a theory can be strengthened at will, in order to find answers to Max-SMT problems. The paper uses the DPLL framework to prove their variant correct, which allows them to extend an existing DPLL solver with little effort. The resulting solver can handle a well-known "hard" problem, radio frequency assignment, which is not typically thought of as an SMT problem since it requires optimization and not only satisfiability.

The main strength of the paper is that it explains DPLL(T) and the authors' proposed strengthening very well. The contribution is not just useful but also simple enough to understand, and the resulting solver has comparable performance to more complex solvers.

However, the empirical evaluation leaves much to be desired, as the exact benchmarks used are not specified and the results are not always in the new solver's favor. The main evaluation outcome is that the solver is "good enough" compared to more complex ones, but it is entirely possible that with different sets of benchmarks the results could be very different.

We plan on using Max-SMT in order to improve the counter-example generation of a theorem prover such as Stainless. Stainless can use Z3 [3], which implements optimization strategies [1] similar to those described in this paper. Thus, we should be able to minimize counter-examples by finding a function that maps an example to its complexity (which is somewhat subjective) and asking Z3 to minimize the function's output on the example.

References

[1] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νZ - An Optimizing SMT Solver. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 194–199, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[2] Bertrand Cabon, Simon De Givry, Lionel Lobjois, Thomas Schiex, and Joost P. Warners. Radio link frequency assignment. *Constraints*, 4(1):79–89, 1999.

[3] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[4] Niklas Eén and Niklas Sörensson. Translating Pseudo-Boolean Constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation 2:1–25, 2006.

[5] Federico Heras, Simon De Givry, Matthias Zytnicki, and Javier Larrosa. Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In *IJCAI 2005 - International Joint Conference on Artificial Intelligence*, IJCAI'05 Proceedings of the 19th international joint conference on Artificial intelligence, Edimbourg, United Kingdom, July 2005. Morgan Kaufmann.

[6] Robert Nieuwenhuis and Albert Oliveras. On SAT modulo Theories and Optimization Problems. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, SAT'06, page 156–169, Berlin, Heidelberg, 2006. Springer-Verlag.

[7] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *J. ACM*, 53(6):937–977, November 2006.

[8] Solal Pirelli and Matthieu Bovel. Toy dpll implementations. https://github.com/mbovel/formal-verification-project/tree/main/paper-test-implementation, 2021.

[9] Hossein M. Sheini and Karem A. Sakallah. Pueblo: A Modern Pseudo-Boolean SAT Solver. In EDAA European design and Automation Association, editors, *DATE'05*, volume 2, pages 684–685, Munich, Germany, March 2005. Submitted on behalf of EDAA (http://www.edaa.com/).