SAT and SMT algorithms[1]

Paul Jackson
Paul.Jackson@ed.ac.uk

School of Informatics
University of Edinburgh

Formal Verification
Autumn 2023

[1]Including contributions by Elizabeth Polgreen

Basic questions

SAT: Given a propositional logic formula, is it satisfiable?

SMT (SAT Modulo Theories): Given a logical formula over theories (e.g. $\mathbb{Z}$, $\mathbb{R}$, arrays, uninterpreted functions), is it satisfiable?

- A formula is *valid* just when its negation is*unsatisfiable*
- Hence SAT & SMT solvers are also automatic theorem provers

Huge range of applications

- Reasoning engines for many kinds of formal verification tools
- Constraint Programming: e.g. planning, scheduling, Suduko
- Automatic test-case generation for programs
- Synthesis of programs & systems from specifications

SAT solver progress

SAT is NP-complete: no polynomial time algorithm.

Yet, huge progress has been made in size of formula that modern SAT solvers can solve:

Year	# Vars
1960	80
1970	100
1980	120
1990	700
2000	3,000
2010	600,000

Size of realistic problems solved in a few hours

Terminology

- An atom p is a propositional symbol

  Also call an atom a propositional variable or simply a variable.

- A literal l is an atom p or the negation of an atom $\neg p$.

- A clause C is a disjunction of literals $l_1 \vee \ldots \vee l_n$.

- A CNF formula F is a conjunction of clauses $C_1 \wedge \ldots \wedge C_m$

  (CNF $\equiv$ Conjunctive Normal Form)

Use of CNF

Standard to always first convert formulas to CNF

- Can get exponential blow-up in size.

  Consider putting into CNF

$$(x_1 \wedge x_2) \vee \ldots \vee (x_{2n} \wedge x_{2n+1})$$

- If introduce a new variable for each non-terminal in a formula's syntax tree, can get an equi-satisfiable formula with constant-factor growth in formula size (Tseitin's encoding)

$$x_1 \Rightarrow (x_2 \wedge x_3)$$

  becomes, with new variables z_1 and z_2,

$$z_1 \wedge (z_1 \Leftrightarrow (x_1 \Rightarrow z_2)) \wedge (z_2 \Leftrightarrow ((x_2 \wedge x_3))$$

  which can easily be converted to CNF with a constant growth-factor

Abstract rules for DPLL

Core algorithms used in SAT and SMT solvers derived from DPLL algorithm (Davis,Putnam,Logemann,Loveland) from 1962.

Here present algorithms using abstract rule-based system due to Nieuwenhuis, Oliveras and Tinelli.

- General structure of algorithms easy to see
- Can work through simple examples on paper

General approach

- Try to incrementally build a satisfying truth assignment M for a CNF formula F
- Grow M by
  - guessing truth value of a literal not assigned in M
  - deducing truth value from current M and F.
- If reach a contradiction ($M \models \neg C$ for some $C \in F$), undo some assignments in M and try starting to grow M again in a different way.
- If all variables from M assigned and no contradiction, a satisfying assignment has been found for F
- If exhaust possibilities for M and no satisfying assignment is found, F is unsatisfiable

Assignments and States

States:

$$\textbf{fail} \quad \text{or} \quad M \;\|\; F$$

where

- M is sequence of literals and decision points $\bullet$ denoting a partial truth assignment
- F is a set of clauses denoting a CNF formula

First literal after each $\bullet$ is called a decision literal

Decision points start suffixes of M that might be discarded when choosing new search direction

Def: If $M = M_0 \bullet M_1 \bullet \cdots \bullet M_n$ where each M_i contains no decision points

- M_i is decision level i of M
- M_n, decision level n, is the current decision level

Initial and final states

Initial state

- $() \parallel F_0$

Expected final states

- **fail** if F_0 is unsatisfiable
- $M \parallel G$ otherwise, where
  - G is equivalent to F_0
  - M satisfies G

Classic DPLL rules

Decide

$$M \parallel F \Longrightarrow M \bullet l \parallel F \quad \textbf{if} \begin{cases} l \text{ or } \neg l \text{ in clause of } F, \\ l \text{ is undefined in } M \end{cases}$$

Backtrack

$$M \bullet l\ N \parallel F, C \Longrightarrow M\ \neg l \parallel F, C \quad \textbf{if} \begin{cases} M \bullet l\ N \models \neg C \\ \bullet \notin N \end{cases}$$

Fail

$$M \parallel F, C \Longrightarrow \textbf{fail} \quad \textbf{if} \begin{cases} M \models \neg C, \\ \bullet \notin M \end{cases}$$

UnitPropagate

$$M \parallel F, C \vee l \Longrightarrow M\ l \parallel F, C \vee l \quad \textbf{if} \begin{cases} M \models \neg C, \\ l \text{ is undefined in } M \end{cases}$$

Strategies for applying rules

- After each Decide or UnitPropagate should check for a conflicting clause, a clause C for which

$$M \models \neg C \quad .$$

  If there is a conflicting clause, Backtrack or Fail are applied immediately to avoid pointless search.

- UnitPropagate applied with higher priority than Decide since it does not introduce branching in search
  - Typically many UnitPropagate applications for each Decide
  - BCP (Boolean Constraint Propagation): repeated application of UnitPropagate

Strategies for applying rules (cont)

- Are many heuristics for choosing literal l in Decide rule.
  - DLIS (Dynamic Largest Individual Sums): choose the unassigned literal that satisfies the largest number of currently unsatisfied clauses
  - MOMS: choose literal with the Maximum number of Occurrences in Minimum Size clauses.
  - VSIDS (Variable State Independent Decaying Sum): choose literal that has most frequently been involved in recent conflict clauses.

Heuristics striving for choice with maximum impact

Example execution

	C_1		C_2		C_3		C_4			
M	$\bar{x}_1 \vee$	x_2	$\bar{x}_3 \vee$	x_4	$\bar{x}_5 \vee$	$\bar{x}_6$	$x_6 \vee$	$\bar{x}_5 \vee$	$\bar{x}_2$	Rule
$()$	u	u	u	u	u	u	u	u	u	
$\bullet x_1$	$\underline{0}$	$\underline{u}$	u	u	u	u	u	u	u	Decide x_1
$\bullet x_1 x_2$	0	1	u	u	u	u	u	u	0	UnitProp C_1
$\bullet x_1 x_2 \bullet x_3$	0	1	$\underline{0}$	$\underline{u}$	u	u	u	u	0	Decide x_3
$\bullet x_1 x_2 \bullet x_3 x_4$	0	1	0	1	u	u	u	u	0	UnitProp C_2
$\bullet x_1 x_2 \bullet x_3 x_4 \bullet x_5$	0	1	0	1	$\underline{0}$	$\underline{u}$	u	0	0	Decide x_5
$\bullet x_1 x_2 \bullet x_3 x_4 \bullet x_5 \bar{x}_6$	0	1	0	1	0	1	$\underline{0}$	$\underline{0}$	$\underline{0}$	UnitProp C_3
$\bullet x_1 x_2 \bullet x_3 x_4 \bar{x}_5$	0	1	0	1	1	u	u	1	0	Backtrack C_4
$\bullet x_1 x_2 \bullet x_3 x_4 \bar{x}_5 \bar{x}_6$	0	1	0	1	1	1	0	1	0	Decide $\bar{x}_6$

- Last state here is final – no further rules apply
- Derivation shows that $C_1 \wedge C_2 \wedge C_3 \wedge C_4$ is satisfiable
- Final M is a satisfying assignment

Implication graphs

An implication graph describes the dependencies between literals in an assignment

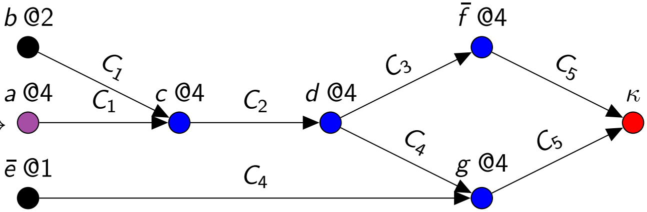
- 1 node per assigned literal
  - Node label $l\,@i$ indicates literal l is assigned true at decision level i.
- Roots of graph (nodes without in-edges) are literals in M_0 and decision literals
- l in-edges $l_1 \rightarrow l, \cdots, l_n \rightarrow l$ added if unit propagation with clause $\neg l_1 \vee \cdots \vee \neg l_n \vee l$ sets literal l
  - Each edge labelled with clause
  - Edges indicate that $(l_1 \wedge \cdots \wedge l_n) \Rightarrow l$
- When current assignment is conflicting with conflicting clause $\neg l_1 \vee \cdots \vee \neg l_n$, then conflict node κ and κ in-edges $l_1 \rightarrow \kappa, \cdots, l_n \rightarrow \kappa$ added
  - Each edge labelled with conflicting clause
  - Edges indicate that $(l_1 \wedge \cdots \wedge l_n) \Rightarrow$ **false**

Partial Implication graph example

Only shows current decision-level nodes and immediately-preceding nodes.

$$
\begin{array}{lll}
C_1 = \bar{a} \vee \bar{b} \vee c & C_2 = \bar{c} \vee d & C_3 = \bar{d} \vee \bar{f} \\
C_4 = \bar{d} \vee e \vee g & C_5 = f \vee \bar{g} &
\end{array}
$$

Decision literal →



Backjump clause inference

The implication graph enables inference of new clauses that are

1. entailed by the current formula F, and
2. conflicting clauses under the current assignment.

- Consider any cut of an implication graph with
  - On right: conflicting node κ
  - On left: decision literal for current level and all literals at lower levels
- If literals on immediate left of cut are $l_1, \ldots, l_n$, then can infer the new clause

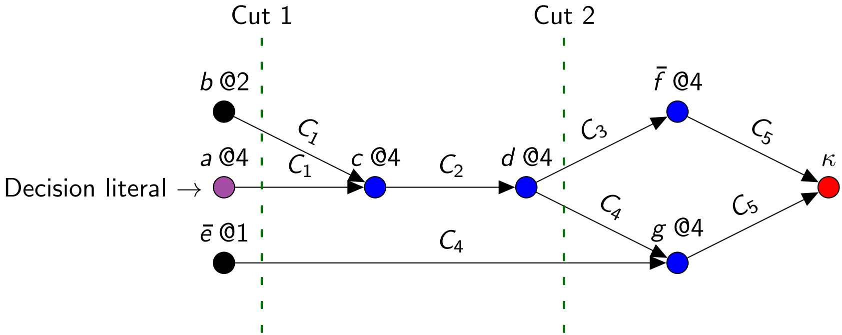
  $$(l_1 \wedge \cdots \wedge l_n) \Rightarrow \textbf{false}$$

  or equivalently

  $$\neg l_1 \vee \cdots \vee \neg l_n$$

Clause inference example

$$C_1 = \bar{a} \vee \bar{b} \vee c \quad C_2 = \bar{c} \vee d \quad C_3 = \bar{d} \vee \bar{f}$$
$$C_4 = \bar{d} \vee e \vee g \quad C_5 = f \vee \bar{g}$$



Backjump clause: $\bar{b} \vee \bar{a} \vee e$ $\bar{d} \vee e$

Backjumping

If

- current assignment has form $M \bullet l\ N$,
- there is some conflicting clause under this assignment,
- an inferred clause has form $C' \vee l'$ where l' is the only literal at the current decision level,
- all literals of C' are assigned in M,

then it is legitimate to

- backjump, set the assignment to M, and
- noting that $C' \vee l'$ has exactly one literal unassigned in M, to apply unit propagation to extend the assignment to $M\ l'$.

The clause $C' \vee l'$ is called a backjump clause and the literal l' is called a unique implication point (UIP).

- One UIP is the decision literal from the current level
- More generally, a UIP is any literal at the current level that appears on every path from from the current decision literal to the conflict node κ.
- Often the UIP closest to κ is chosen

Backjump rule

Replaces and generalises Backtrack rule in modern DPLL implementations

Backjump

$$M \bullet l\ N \ \| \ F, C \Longrightarrow M\ l' \ \| \ F, C \ \ \textbf{if} \begin{cases} M \bullet l\ N \models \neg C, \text{ and there is some clause } C' \vee l' \text{ such that:} \\ -\ F, C \models C' \vee l', \\ -\ M \models \neg C', \\ -\ l' \text{ is undefined in } M, \text{ and} \\ -\ l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } M \bullet l\ N \end{cases}$$

- C is the conflicting clause
- $C' \vee l'$ is the backjump clause

Learning

Learn

$$M \parallel F \Longrightarrow M \parallel F, C \quad \textbf{if} \begin{cases} \text{each atom of } C \text{ occurs in} \\ F \text{ or in } M, \\ F \models C \end{cases}$$

- Common C are backjump clauses from the Backjump rule.
- Learned clauses record information about parts of search space to be avoided in future search
- CDCL (Conflict Driven Clause Learning)
  $=$ Backjump $+$ Learn

Forgetting

Forget

$$M \parallel F, C \Longrightarrow M \parallel F \;\; \textbf{if } F \models C$$

- Applied to C considered less important.
- Essential for controlling growth of required storage.
- Performance can degrade as F grows, so shrinking F can improve performance.

Restarting

Restart

$$M \parallel F \Longrightarrow () \parallel F$$

- Only used if F grown using learning.
- Additional knowledge causes Decide heuristics to work differently and often explore search space in more compact way.
- To preserve completeness, applied repeatedly with increasing periodicity.

Why is DPLL correct? 1

Lemma (1 - nature of reachable states)

Assume $() \parallel F \Longrightarrow^* M \parallel F'$. *then*

1. F *and* F' *are equivalent*
2. *If* M *is of the form* $M_0 \bullet l_1 M_1 \cdots \bullet l_n M_n$ *where all* M_i *are* $\bullet$ *free, then* $F, l_1, \ldots l_i \models M_i$ *for all* i *in* $0 \ldots n$.

Lemma (2 - nature of final states)

If $() \parallel F \Longrightarrow^* S$ *and* S *is final (no further transitions possible), then either*

1. $S = \textbf{fail}$, *or*
2. $S = M \parallel F'$ *where* $M \models F$

Why is DPLL correct? 2

Lemma (3 - transition sequences never go on for ever)

Every derivation $() \parallel F \Longrightarrow S_1 \Longrightarrow S_2 \Longrightarrow \cdots$ *is finite*

Proof.

Given M of form $M_0 \bullet M_1 \cdots \bullet M_n$ where all M_i are $\bullet$ free, define the *rank of* M, $\rho(M)$ as $\langle r_0, r_1, \ldots, r_n \rangle$ where $r_i = |M_i|$. Every derivation must be finite as each basic DPLL rule strictly increases the rank in a lexicographic order and the image of ρ is finite. □

Why is DPLL correct? 3

Theorem (1 - termination in **fail** state)

If $() \parallel F \Longrightarrow^* S$ *and* S *is final, then*

1. *if* S *is* **fail**, *then* F *is unsatisfiable*
2. *if* F *is unsatisfiable then* S *is* **fail**

Why is DPLL correct? 4

Proof.

1. We have $() \parallel F \Longrightarrow^* M \parallel F' \Longrightarrow$ **fail**.

   By Fail rule definition, there is a $C \in F'$ s.t. $M \models \neg C$.

   Since M is $\bullet$ free, we have by Lemma 1(2) that $F \models M$, and therefore $F \models \neg C$.

   However, $F' \models C$ and by Lemma 1(1) $F \models C$.

   Hence, F must be unsatisfiable.

2. By Lemmas 2 and 3.

$\square$

Abstract DPLL modulo theories

Start just with one theory T. E.g.

- Equality with uninterpreted functions
- Linear arithmetic over $\mathbb{Z}$ or $\mathbb{R}$.

Propositional atoms now both

- Propositional symbols
- Atomic relations over T involving individual expressions.
  E.g. $f(g(a)) = b$ or $3a + 5b \leq 7$.

Previous rules (e.g. Decide, UnitPropagate) and $\models$ (propositional entailment) treat syntactically distinct atoms as distinct

New rules involve $\models_T$ (entailment in theory T)

$\models_T$ is more general.
E.g. $\models_T x \leq 2 \vee x \geq 1$ but $\not\models x \leq 2 \vee x \geq 1$

Theory learning

T-Learn

$$M \parallel F \Longrightarrow M \parallel F, C \quad \textbf{if} \begin{cases} \text{each atom of } C \text{ occurs in} \\ F \text{ or in } M, \\ F \models_T C \end{cases}$$

- One use is for catching when M is inconsistent from T point of view.
  - Say $\{l_1, \ldots, l_n\} \subseteq M$ such that $F \models_T (l_1 \wedge \cdots \wedge l_n) \Rightarrow$ **false**
  - Then add $C = \neg l_1 \vee \cdots \vee \neg l_n$
  - As C is conflicting, the Backjump or Fail rule is enabled
  - Theory solvers can identify unsat cores, small subsets of literals sufficient for creating a conflicting clause
- Frequency of checks $F \models_T C$ needs careful regulation, as cost might be far higher than basic DPLL steps.
- Given size of F often just check $\models_T C$. In this case C is called a theory lemma.

Theory propagation

Guiding growth of M rather than just detecting when it is T-inconsistent.

TheoryPropagate

$$M \parallel F \Longrightarrow M\, l \parallel F \quad \textbf{if} \begin{cases} M \models_T l, \\ l \text{ or } \neg l \text{ occurs in } F \\ l \text{ is undefined in } M \end{cases}$$

- If applied well, can dramatically increase performance
- Worth applying exhaustively in some cases before resorting to Decide

Integration of SAT and theory solvers

Further new rules T-Backjump and T-Forget which generalise Backjump and Forget are also needed.

Use of theory-sensitive rules rules requires close integration of SAT and theory solvers

- SAT solvers need modification to be able to call out to theory solvers
- Useful to have theory solvers incremental, able to be rerun efficiently when input is some small increment on previous input
  - Also theory solvers need to support efficient retraction of blocks of input to cope with backjumping

Handling multiple theories

Consider formula F mixing theories of linear real arithmetic and uninterpreted functions:

$$\begin{gathered} f(x_1, 0) \geq x_3 \ \wedge \ f(x_2, 0) \leq x_3 \wedge \\ x_1 \geq x_2 \ \wedge \ x_2 \geq x_2 \wedge \\ x_3 - f(x_1, 0) \geq 1 \end{gathered}$$

The popular Nelson-Oppen combination procedure involves first purifying, adding additional variables and creating an equisatisfiable formula with each atom over just one of the theories.

Formula F above is equisatisfiable with $F_1 \wedge F_2$, where

$$\begin{array}{rcl} F_1 & = & a_1 \geq x_3 \ \wedge \ a_2 \leq x_3 \ \wedge \ x_1 \geq x_2 \ \wedge \ x_2 \geq x_1 \wedge \\ & & x_3 - a_1 \geq 1 \ \wedge \ a_0 = 0 \\ F_2 & = & a_1 = f(x_1, a_0) \ \wedge \ a_2 = f(x_2, a_0) \end{array}$$

F_1 just involves linear real arithmetic and F_2 just involves an uninterpreted function

Nelson-Oppen example

Separate theory solvers can work on F_1 and F_2, exchanging equalities

i	1	2
	R arith	EUF
Original F_i	$a_1 \geq x_3$	$a_1 = f(x_1, a_0)$
	$a_2 \leq x_3$	$a_2 = f(x_2, a_0)$
	$x_1 \geq x_2$	
	$x_2 \geq x_1$	
	$x_3 - a_1 \geq 1$	
	$a_0 = 0$	
Deduced	$x_1 = x_2 (*)$	$x_1 = x_2$
atoms	$a_1 = a_2$	$a_1 = a_2 (*)$
	$a_1 = x_3 (*)$	
	false$(*)$	

The $(*)$ marks indicate when inference is in the respective theory

Nelson-Oppen

The basic Nelson-Oppen procedure relies on each theory T being combined being convex:

> For any set of literals L, if $L \models_T s_1 = t_1 \vee \cdots \vee s_n = t_n$ then $L \models_T s_i = t_i$ for some i.

- Linear real arithmetic and EUF (Equality and Uninterpreted Functions) are convex.
- Linear integer arithmetic and bit-vector theories are not.

  If L is $\{0 \leq x, x \leq 1\}$, then $L \models_{\mathbb{Z}} x = 0 \vee x = 1$, but $L \not\models_{\mathbb{Z}} x = 0$ and $L \not\models_{\mathbb{Z}} x = 1$

Extensions of Nelson-Oppen can handle a number of non-convex theories.

In general, a combination of decidable theories might be undecidable

Further reading

1. *Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T)* Robert Neiuwenhuis, Albert Oliveras, Cesare Tinelli. Journal of the ACM. 53(6):937-977, 2006

   Main source for Abstract DPLL approach adopted in slides

2. Slides and videos from the 2012 SAT/SMT Summer School

   Tinelli's presentation uses refined version of Abstract DPLL

3. SAT/SMT/AR/CP Summer Schools, 2011-2022

   See later schools for an introduction to recent work and applications.

4. *Decision Procedures: An Algorithmic Point of View*. D Kroening, O. Strichman. 2nd Ed. 2016. Springer Nature. Online from Learn Resource List.

   Additional source for slides. Does not do Abstract DPLL. Good reference for recent work.