SAT/SMT Solvers and Their Applications

Ashutosh Gupta

TIFR, India

Compile date: 2017-03-01

Logic is the backbone of formal methods

Differential equations
are the calculus of
Electrical engineering

Logic
is the calculus of
Formal methods

Logic provides tools to define/manipulate computational objects

Topic 1.1

SAT problem

Example: SAT problem

Let x, y be rational variables.

Choose a value of x and y such that the following formula holds true.

$$x + y = 3$$

We say

$$\{x \mapsto 1, y \mapsto 2\} \models x + y = 3$$

Commentary: We are not calling x and y rational numbers. They are not numbers. They are symbols that can hold numbers.

Example: SAT problem(contd.)

Let x, y be rational variables.

Choose a value of x and y such that the following formula holds true.

$x + y = 3 \wedge y > 10 \wedge x > 0$ theory formulas

$x + y = 3 \wedge y > 10 \wedge (x > 0 \vee x < -4)$ Quantifier-free

$\forall y.\ x + y = 3 \wedge y > 10 \wedge (x > 0 \vee x < -4)$ quantified formulas

Easy

Hard

Impossible

Commentary: The above are increasingly harder class of satisfiability problems.

Solvers for Quantifier-free formulas

We will look at satisfiability solvers for the quantifier-free formulas that consists of

- Theory atoms
- Boolean structure

Example 1.1

$$x + y = 3 \wedge y > 10 \wedge (x > 0 \vee x < -4)$$

Theory atoms

Boolean Structure

A comment on theories

Theory is a technical name for the subject of interest.

- Rationals
- Integers
- Reals
- Floats
- Arrays
- Chairs
- Cartoons

Theory is a very general concept.

Let us stick to rational/integer arithmetic in this talk.

Propositional formulas

Propositional formulas are a special case, where the theory atoms are Boolean variables.

Example 1.2

Let p_1, p_2, p_3 be Boolean variables.

$$p_1 \wedge \neg p_2 \wedge (p_3 \vee p_2)$$

A satisfying assignment.

$$\{p_1 \mapsto 1, p_2 \mapsto 0, p_3 \mapsto 1\} \models p_1 \wedge \neg p_2 \wedge (p_3 \vee p_2)$$

A bit of jargon

- Solvers for quantifier-free propositional formulas are called

SAT solvers.

- Solvers for quantifier-free formulas with the other theories are called

SMT solvers.

SMT = satisfiability modulo theory

Topic 1.2

SAT problems are everywhere

SAT problems

Every field of S&T encounters SAT problem of quantifier-free formulas.

A few are listed here

- Hardware verification and design assistance
  Almost all hardware/EDA companies have their own SAT solver
- Planning: many resource allocation problems are convertible to SAT
- Security: analysis of crypto algorithms
- Solving hard problems, *e. g.*, travelling salesman problem
- Sampling/counting

Example: Solving Sudoku using SAT solvers

Example 1.3

- *Variables:* $v_{i,j,k} \in \mathcal{B}$ *and* $i, j, k \in \{1, ..., 9\}$
- *If* $v_{i,j,k} = 1$, *column* i *and row* j *contains* k.
- *Value in each cell is valid:*

$$\sum_{k=1}^{9} v_{i,j,k} = 1 \qquad i, j \in \{1, .., 9\}$$

- *Each value used exactly once in each row:*

$$\sum_{i=1}^{9} v_{i,j,k} = 1 \qquad j, k \in \{1, .., 9\}$$

- *Each value used exactly once in each column:*

$$\sum_{j=1}^{9} v_{i,j,k} = 1 \qquad i, k \in \{1, .., 9\}$$

- *Each value used exactly once in each* 3×3 *grid*

$$\sum_{s=1}^{3} \sum_{r=1}^{3} v_{3i+r,j+s,k} = 1 \quad i, j \in \{0, 1, 2\}, k \in \{1, .., 9\}$$

Encoding $x_1 + ... + x_k = 1$

- At least one of x_i is true

$$(x_1 \vee .... \vee x_k)$$

- Not more than two x_is are true

$$(\neg x_i \vee \neg x_j) \qquad i, j \in \{1, .., 9\}$$

SMT problem in bug detection

Example 1.4

Consider program

```
foo(x,y) {
  u=x+y;
  if (u!=1)
    z=2;
  else
    z=u+1;
  u = y/z;//avoid divide by 0
  return u;
}
```

The following formula in quantifier-free linear integer arithmetic encodes the program behaviors

$$u = x + y \wedge$$
$$(u = 1 \wedge z = 2 \vee u \neq 1 \wedge z = u + 1) \wedge$$
$$z = 0$$

If the above formula is sat, the program has a bug

Detailed presentation will be given on Tuesday

Topic 1.3

Rise of Solvers

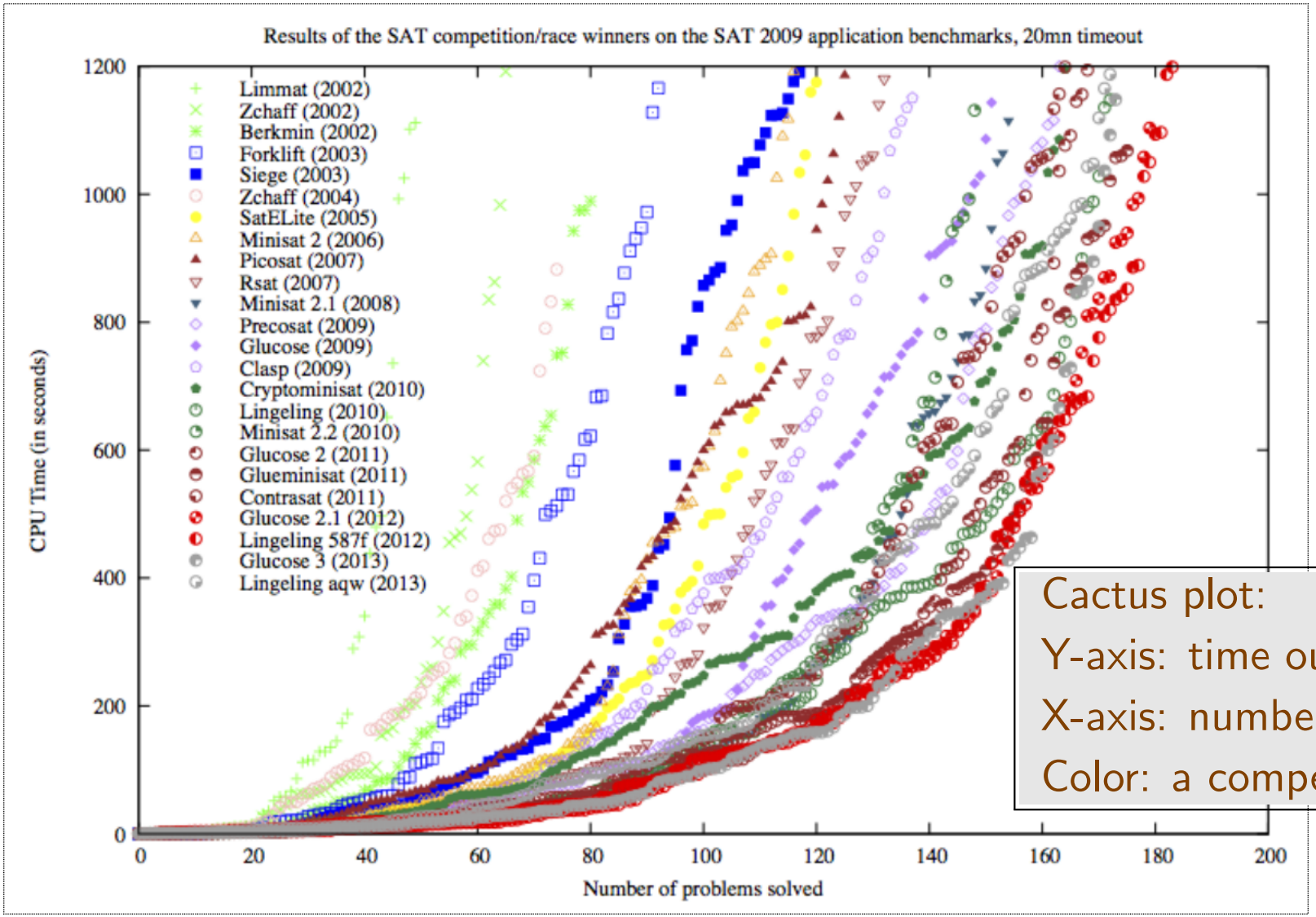
Rise of SAT/SMT solvers

SAT solving is theoretically known to be a hard problem.

However, it did not stop researchers to attempt building practical solvers.

- In early 2000s, stable and scalable SAT/SMT solvers started appearing. e.g., zChaff, Yiecs
- SAT/SMT competitions became a driving force in their ever increasing efficiency
- Formal methods community quickly realized their potential
- Z3, one of the leading SMT solver, alone has about 3000+ citations (375 per year)(June 2016)

Efficiency of SAT solvers over the years



Cactus plot:

Y-axis: time out

X-axis: number of solved probl

Color: a competing solver

Source: `http://satsmt2014.forsyte.at/files/2014/07/SAT-introduction.pdf`

SAT technology: quite revolution

Impact is enormous.

Probably, one of the greatest achievement of the first decade of this century

All verification tools depends on the solvers.

Topic 1.4

SAT solver

Some terminology

- Propositional variables are also referred as atoms
- A literal is either an atom or its negation
- A clause is a disjunction of literals.
- A formula is in CNF if it is a conjunction of clauses.

Example 1.5

- *p is an atom but $\neg p$ is not.*
- *$\neg p$ and p both are literals.*
- *$p \vee \neg p \vee p \vee q$ is a clause.*
- *$\neg p$ and p both are in CNF.*
- *$(p \vee \neg q) \wedge (r \vee \neg q) \wedge \neg r$ is in CNF.*
- *$(p \vee \neg q) \wedge ((r \wedge \neg p) \vee \neg q) \wedge \neg r$ is not in CNF.*

Definition 1.1

Let $atoms(F)$ denote the set of atoms appearing in F.

Partial model

Definition 1.2

For a CNF F, A partial model m is an ordered partial map from atoms(F) to $\mathcal{B}$.

Example 1.6

partial models $m_1 = \{x \mapsto 0, y \mapsto 1\}$ and $m_2 = \{y \mapsto 1, x \mapsto 0\}$ are not same.

Some notation

Before presenting the solvers, let us define some notations.

Under partial model m,

A literal ℓ is true if $m(\ell) = 1$ and
ℓ is false if $m(\ell) = 0$.
Otherwise, ℓ is undefined.

A clause C is true if there is $\ell \in C$ s.t. ℓ is true and
C is false if for each $\ell \in C$, ℓ is false.
Otherwise, C is undefined.

CNF F is true if for each $C \in F$, C is true and
F is false if there is $C \in F$ s.t. C is false.
Otherwise, F is undefined.

Unit clause and unit literal

Definition 1.3

C is a unit clause under m if a literal $\ell \in C$ is undefined and the rest are false. ℓ is called unit literal.

DPLL (Davis-Putnam-Loveland-Logemann)

Algorithm 1.1: DPLL(F,m)

Input: CNF F, partial model m

if *F is true under m* **then**
  return *sat*

if *F is false under m* **then**
  return *unsat*

Backtracking at conflict

if *$\exists$ unit literal x under m* **then**
  return $DPLL(F, m[x \mapsto 1])$

if *$\exists$ unit literal $\neg x$ under m* **then**
  return $DPLL(F, m[x \mapsto 0])$

Choose an undefined x;

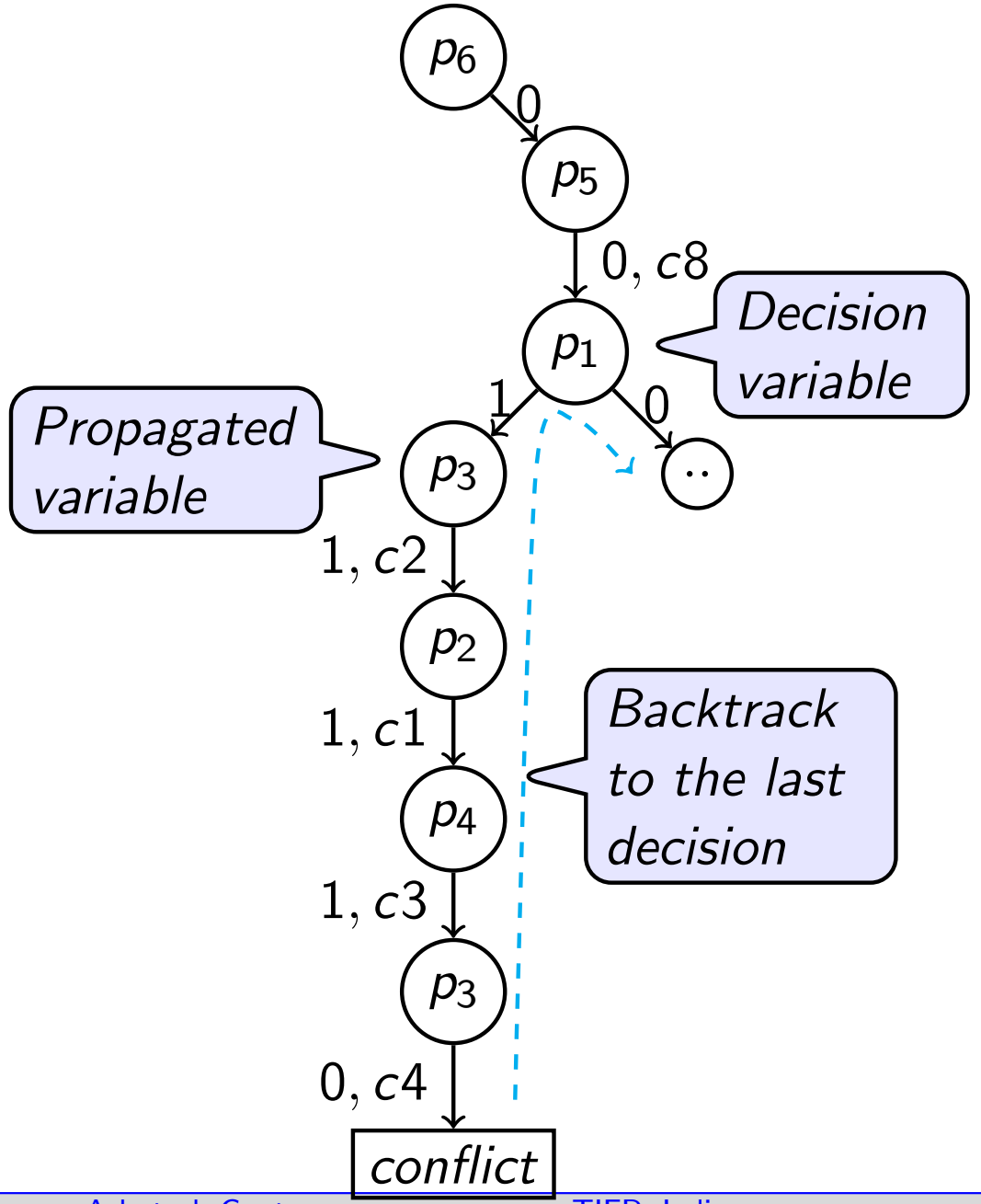
if $DPLL(F, m[x \mapsto 0]) == sat$ **then**
  return *sat*

else
  return $DPLL(F, m[x \mapsto 1])$

Example: Brancing and bracktracking in DPLL

Example 1.7

$$c_1 = (\neg p_1 \vee p_2)$$
$$c_2 = (\neg p_1 \vee p_3 \vee p_5)$$
$$c_3 = (\neg p_2 \vee p_4)$$
$$c_4 = (\neg p_3 \vee \neg p_4)$$
$$c_5 = (p_1 \vee p_5 \vee \neg p_2)$$
$$c_6 = (p_2 \vee p_3)$$
$$c_7 = (p_2 \vee \neg p_3)$$
$$c_8 = (p_6 \vee \neg p_5)$$



Exercise 1.1

Complete the DPLL run

Optimizations

There are various optimizations in implementing DPLL

We will discuss only four optimizations.

- **clause learning**
- 2-watched literals
- variable ordering
- restarts

Topic 1.5

Clause learning

Clause learning

As we decide and propagate, we may construct a data structure that allows us to do efficient back tracking.

Definition 1.4 (implication graph)

An implication graph is a labeled directed graph (N, E)*, where*

- N *contains true literals and a conflict node to denote contradiction*
- $E = \{(\ell_1, \ell_2) | \neg\ell_1 \in \mathit{clause}(\ell_2)\}$

$\mathit{clause}(\ell) \triangleq$ *clause due to which unit propagation made* ℓ *true*

Note: For decision literals $\mathit{clause}(\ell)$ *is undefined*

Note: Not same definition as defined for 2-SAT!

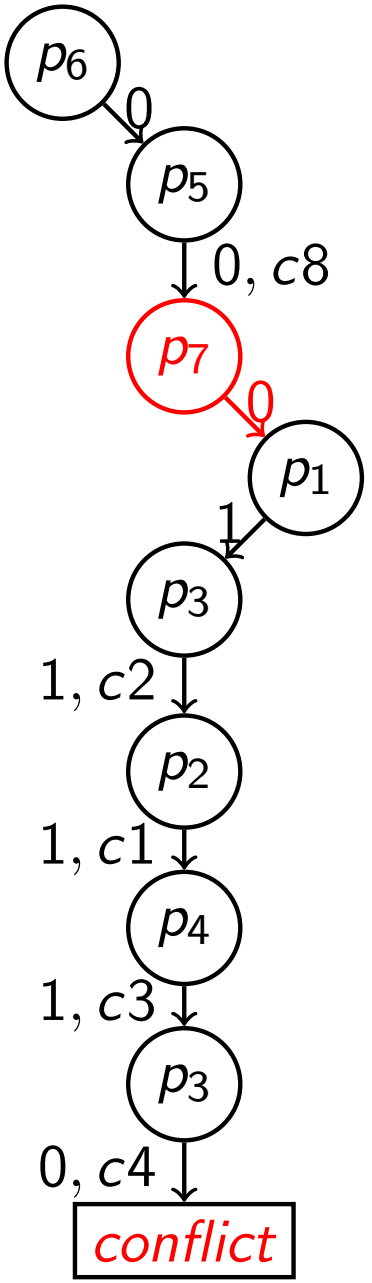
We also annotate each node with decision level (*e. g.*, $\neg p@3$), i.e., the number of decisions after which the variable was assigned

Example: implication graph

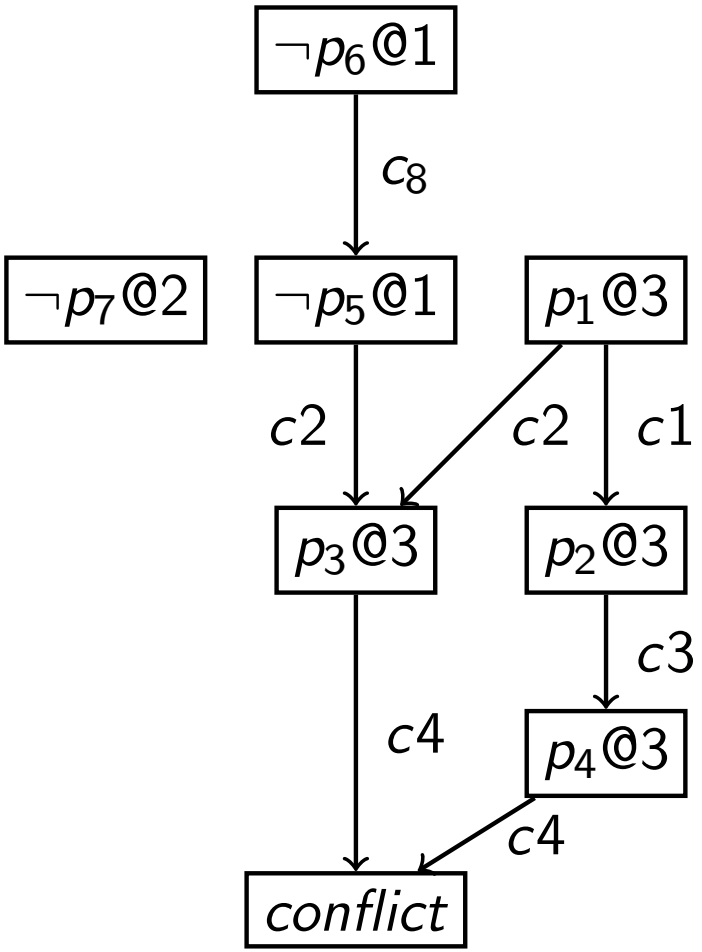
Example 1.8

$$c_1 = (\neg p_1 \vee p_2)$$
$$c_2 = (\neg p_1 \vee p_3 \vee p_5)$$
$$c_3 = (\neg p_2 \vee p_4)$$
$$c_4 = (\neg p_3 \vee \neg p_4)$$
$$c_5 = (p_1 \vee p_5 \vee \neg p_2)$$
$$c_6 = (p_2 \vee p_3)$$
$$c_7 = (p_2 \vee \neg p_3 \vee p_7)$$
$$c_8 = (p_6 \vee \neg p_5)$$

Note: Modified example



Implication graph

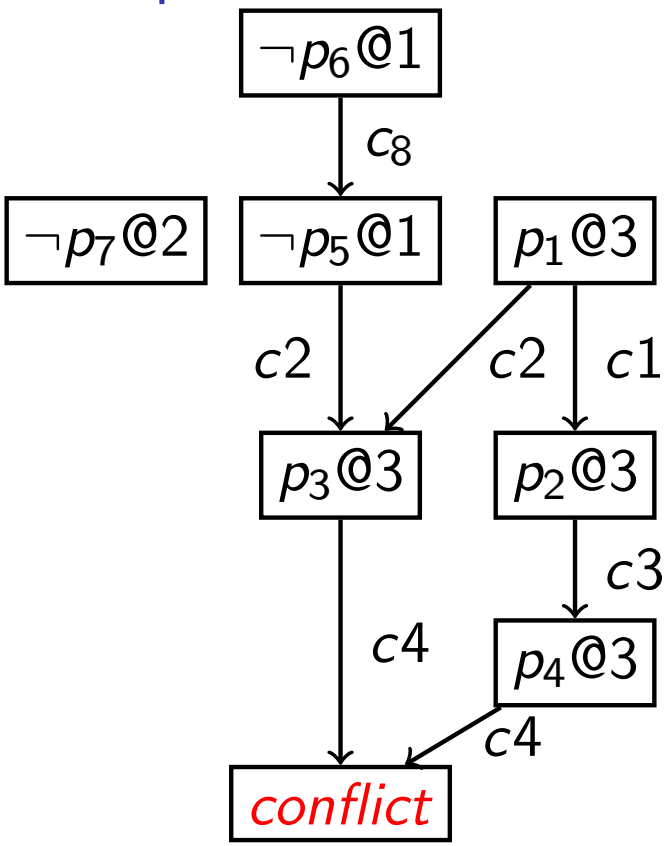


Conflict clause

In the case of conflict, we traverse the implication graph backwards to find the set of decisions that caused the conflict.

The clause of the negations of the decisions is called conflict clause.

Example 1.9



Conflict clause : $p_6 \vee \neg p_1$

Clause learning

Clause learning heuristics

- add conflict clause in the input clauses and
- backtrack to the second last conflicting decision, and proceed like DPLL

Benefit of adding conflict clauses

1. Prunes away search space
2. Records past work of the SAT solver
3. Enables very many other heuristics without much complications. We will see them shortly.

Example 1.10

In the previous example, we made decisions :
$m(p_6) = 0$, $m(p_7) = 0$, *and* $m(p_1) = 1$

We learned a conflict clause : $p_6 \vee \neg p_1$

Adding this clause to the input clauses results in

1. $m(p_6) = 0$, $m(p_7) = 1$, *and* $m(p_1) = 1$ *will never be tried*
2. $m(p_6) = 0$ *and* $m(p_1) = 1$ *will never occur simultaneously.*

Impact of clause learning was so profound that some people call the optimized algorithm CDCL(conflict driven clause learning) instead of DPLL

CDCL as an algorithm

Algorithm 1.2: CDCL

Input: CNF F

AddClauses(F); m := UnitPropagation(); $dl := 0$; $dstack := \lambda x.0$;

do

  // backtracking

  while $\exists x\ \{x \mapsto 0, x \mapsto 1\} \subseteq m$ **do**

    if $dl = 0$ **then return** *unsat*;

    (C, dl) := AnalyzeConflict(m);

    $m.resize(dstack(dl))$; AddClauses($\{C\}$); m := UnitPropagation();

  // Boolean decision

  if m *is partial* **then**

    $dstack(dl) := m.size()$;

    $dl := dl + 1$; m := Decide(); m := UnitPropagation() ;

while m *is partial or* $\exists x\ \{x \mapsto 0, x \mapsto 1\} \subseteq m$;

return *sat*

stands for decision level

dstack records history for backtracking

- AddClauses(Cs) - adds Cs to the current set of problem clauses
- UnitPropagation() - applies unit propagation and extends m as much as possible
- Decide() - chooses an undefined variable in m and assigns a Boolean value
- AnalyzeConflict() - returns a conflict clause learned using implication graph and a decision level for back tracking

Topic 1.6

Other heuristics

Other heuristics

Now we will discuss the other heuristics that may improve the performance of SAT solvers

- 2-watched literals
- pure literals
- variable ordering
- restarts
- Learned clause deletion
- Cache aware implementation

Commentary: Clause learning is an algorithmic change. The above optimization are clever data structures and implementations.

2-watched literals

This data structure optimizes unit clause propagation

Observation:
To decide if a clause is ready for unit propagation, we need to look at only two literals that are not false

For each clause we choose two literals and we call them watched literals.

In a clause,

- if watched literals are non-false, the clause is not a unit clause
- if any of the two becomes false, we look for another two non-false literals
- If we can not find another two, the clause is a unit clause

Exercise 1.2

Why this scheme may optimize CDCL?

Example: 2-watched literals

Example 1.11

Consider clause $p_1 \vee p_2 \vee \neg p_3 \vee \neg p_4$ *in a formula among other variables and clauses. Let us suppose initially we watch* p_1 *and* p_2 *in the clause.*

* $\triangleq$ *watched literals.*

☺ $\triangleq$ *no work to be done!*

Initially: $p_1^* \vee p_2^* \vee \neg p_3 \vee \neg p_4 \quad m = \{\}$

$\vdots$

Assign $p_1 = 0$*:* $p_1 \vee p_2^* \vee \neg p_3^* \vee \neg p_4 \quad m = \{\ldots, p_1 \mapsto 0\}$

Assign $p_2 = 1$*:* $p_1 \vee p_2^* \vee \neg p_3^* \vee \neg p_4 \quad m = \{\ldots, p_1 \mapsto 0, p_2 \mapsto 1\}$ ☺

Backtrack to p_1*:* $p_1 \vee p_2^* \vee \neg p_3^* \vee \neg p_4 \quad m = \{\ldots\}$ ☺

Assign $p_4 = 1$*:* $p_1 \vee p_2^* \vee \neg p_3^* \vee \neg p_4 \quad m = \{\ldots, p_4 \mapsto 1\}$ ☺

The benefit: often no work to be done!

Topic 1.7

SMT solver

SMT solver

We will now solve quantifier-free formulas in some theory.

Example 1.12

- $f(x) \approx g(h(x, y))$ *is a formula in QF_EUF.*
- $x > 0 \vee y + x \approx 3.5z$ *is a formula in QF_LRA.*

CDCL($\mathcal{T}$)

CDCL solves(i.e. checks satisfiability) quantifier-free propositional formulas

CDCL($\mathcal{T}$) solves quantifier-free formulas in theory $\mathcal{T}$,

- separates the boolean and theory reasoning,
- proceeds like CDCL, and
- needs support of a $\mathcal{T}$-solver $DP_{\mathcal{T}}$, i.e., a decision procedure for conjunction of literals of $\mathcal{T}$

The tools that are build using CDCL($\mathcal{T}$) are called satisfiablity modulo theory solvers (SMT solvers)

Boolean encoder

For a formula F, let boolean encoder e be a partial map from $atoms(F)$ to fresh boolean variables.

For a term t, let $e(t)$ denote the term obtained by replacing each atom a by $e(a)$ if $e(a)$ is defined.

Example 1.13

Let $F = x < 2 \vee (y > 0 \vee x \geq 2)$
and $e = \{x < 2 \mapsto x_1, y > 0 \mapsto x_2\}$
$e(F) = x_1 \vee (x_2 \vee \neg x_1)$

Definition 1.5

For a partial model m of e, let
$e^{-1}(m) \triangleq \{e^{-1}(x) | x \mapsto 1 \in m\} \cup \{\neg e^{-1}(x) | x \mapsto 0 \in m\}$

CDCL($\mathcal{T}$)

Algorithm 1.3: CDCL($\mathcal{T}$)

Input: CNF F, boolean encoder e

AddClauses($e(F)$); m := UnitPropagation(); $dl := 0$; $dstack := \lambda x.0$;

do

  // backtracking

  while $\exists x\ \{x \mapsto 0, x \mapsto 1\} \subseteq m$ **do**

    if $dl = 0$ **then return** *unsat*;

    (C, dl) := AnalyzeConflict(m) ; // clause learning

    $m.resize(dstack(dl))$; AddClauses($\{C\}$); m := UnitPropagation();

  // Boolean decision

  if *m is partial* **then**

    $dstack(dl) := m.size()$;

    $dl := dl + 1$; m := Decide(); m := UnitPropagation() ;

  // Theory propagation

  if $\forall x\ \{x \mapsto 0, x \mapsto 1\} \not\subseteq m$ **then**

    (Cs, dl') := TheoryDeduction($\bigwedge e^{-1}(m)$);

    if $dl' < dl$ **then** $\{dl = dl'$; $m.resize(dstack(dl))$; $\}$;

    AddClauses($e(Cs)$); m := UnitPropagation();

while *m is partial or* $\exists x\ \{x \mapsto 0, x \mapsto 1\} \subseteq m$;

return *sat*

stands for decision level

dstack records history for backtracking

returns a clause set and a decision level

Theory propagation

TheoryDeduction looks at the atoms assigned so far and checks

- if they are mutually unsatisfiable
- if not, are there other literals from F that are implied by the current assignment

Any implementation must comply with the following goals

- Correctness: boolean model is consistent with $\mathcal{T}$
- Termination: unsat partial models are never repeated

TheoryDeduction

TheoryDeduction solves conjunction of literals and returns a set of clauses and a decision level.

$$(Cs, dl') := \text{TheoryDeduction}(\bigwedge e^{-1}(m))$$

Cs may contain the clauses of the form

$$(\bigwedge L) \Rightarrow \ell$$

where $\ell \in lits(F) \cup \{\bot\}$ and $L \subseteq e^{-1}(m)$.

Note: The RHS need not be a single literal

Requirement form TheoryDeduction

The output of TheoryDeduction must satisfy the following conditions

- If $\bigwedge e^{-1}(m)$ is unsat in $\mathcal{T}$ then Cs must contain a clause with $\ell = \bot$.
- if $\bigwedge e^{-1}(m)$ is sat then $dl' = dl$.
  Otherwise, dl' is the decision level immediately after which the unsatisfiablity occurred (clearly stated shortly).

Example : CDCL($\mathcal{T}$)

Consider $F = (x = y \vee y = z) \wedge (y \neq z \vee z = u) \wedge (z = x)$
$e(F) = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge x_4$

After AddClauses($e(F)$); $m :=$ UnitPropagation()
$m = \{x_4 \mapsto 1\}$

After $m :=$ Decide();
$m = \{x_4 \mapsto 1, x_2 \mapsto 0\}$

After $m :=$ UnitPropagation()
$m = \{x_4 \mapsto 1, x_2 \mapsto 0, x_1 \mapsto 1\}$

After $(Cs, dl') :=$ TheoryDeduction($x = y \wedge y \neq z \wedge z = x$)
$Cs = \{x \neq y \vee y = z \vee z \neq x\}, dl' = 0, e(Cs) = \{\neg x_1 \vee x_2 \vee \neg x_4\}$

After AddClauses($e(Cs)$); $m :=$ UnitPropagation()
$m = \{x_4 \mapsto 1, x_2 \mapsto 0, x_1 \mapsto 1, x_1 \mapsto 0\} \leftarrow$ conflict

Topic 1.8

Theory propagation implementation

Theory propagation implementation - Incremental theory solver

Typically, theory propagation is implemented using incremental/online solvers.

Incremental/online solver $DP_{\mathcal{T}}$

- takes input constraints as a sequence of literals,
- maintains a data structure that defines the solver state and satisfiability of constraints seen so far.
- provides a stack like interface
  - push(ℓ) - adds literal ℓ in "constraint store"
  - pop() - removes last pushed literal from the store
  - checkSat() - checks satisfiability of current store
  - unsatCore() - returns the set of literals that caused unsatisfiablity

Note: We assume that push and pop call checkSat() at the end of their execution. Therefore, explicit calls to checkSat() are not necessary. However, practical tools allow users to choose the policy of calling checkSat() - lazy vs. eager

Theory propagation implementation

Algorithm 1.4: TheoryDeduction

Input: Set of literals Ls

Read only input: m partial model, $dstack$ decision depths, dl current decision level

foreach $\ell \in Ls$ **do**
    $DP_{\mathcal{T}}.push(\ell)$

if $DP_{\mathcal{T}}.checkSat() == unsat$ **then**
    $Ls' := DP_{\mathcal{T}}.unsatCore()$; // minimize clause
    $dl' := \max\{dl'' | \exists \ell \in Ls', i.\ dstack(dl'') < i \wedge m[i] = e(\ell) \mapsto \_\}$;
    return $(\{\neg \bigwedge Ls'\}, dl')$
else
    //implied clauses
    $Cs := \emptyset$;
    foreach $\ell \in Lits(F)$ **do**
        $DP_{\mathcal{T}}.push(\neg\ell)$;
        if $DP_{\mathcal{T}}.checkSat() == unsat$ **then**
            $Ls' := DP_{\mathcal{T}}.unsatCore()$; // ℓ is called implied model and $\neg\ell \in Ls'$
            $Cs := Cs \cup \{\neg \bigwedge Ls'\}$;
        $DP_{\mathcal{T}}.pop()$;
    return *(Cs,dl)*

Topic 1.9

Indian research in SAT/SMT solving

Indian research in SAT/SMT solving

Limited research activity in the field in India.

Solvers for verification tools

are like

engines for the cars.

One must learn to build engines if one wants to build cars.

What should we do?

We need to build an eco-system for the field

- funding for the backend research
- concretely defined projects
- more users/researchers interactions
- support for start-ups in the related areas
- no expectations of finished products from academia
- support promising individuals