SAT Solving: 30 Years of CDCL, 20 Years of Proofs, 15 Years of Inprocessing, 3 Years of User Propagator

Mathias Fleury

August 23, 2026
VTSA 2026
University Freiburg / Siemens

Slides with content taken from Mara Besemer, Armin Biere, Florian Pollitt, André Schidler

Introduction

SAT is at the core of:

- low-enough expressiveness to be solvable...
- ... still enough to have applications — AI, planning, bio, crypto, math, ...
- interesting enough that lots and lots of brain-hours have been put in

Other automated reasoning applications:

SMT SAT + more theories (quantifiers, string, BV) — uses SAT internally

Automated Reasoning (First order), not enough applications — competitive until the 2000s

QBF (Boolean quantifier), not enough applications

Planning names are very different, but lots of shared ideas

Introduction

For a long time, the idea was:

Motivation

Translate to your NP-complete problem to SAT, this is better than witing your own

Less clear that before with AI and specialized heuristics

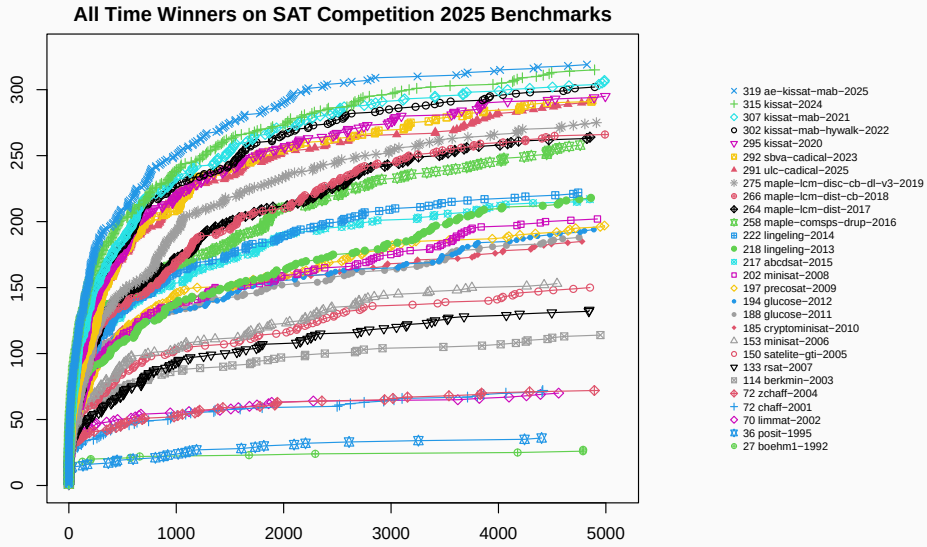
Introduction

For a long time, the idea was:

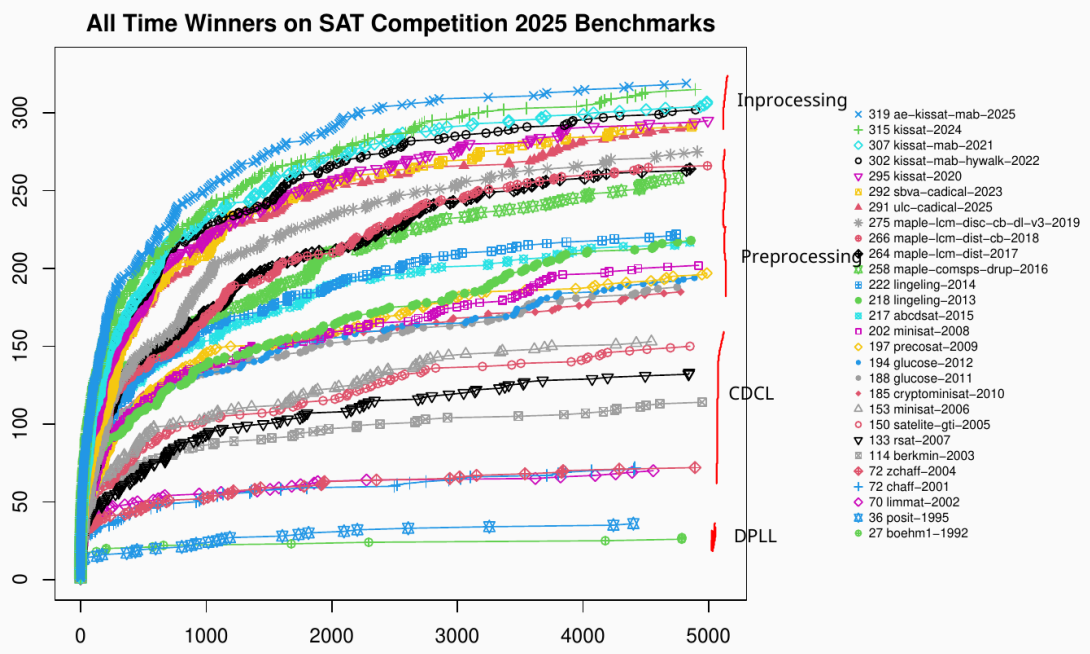
Motivation
Translate to your NP-complete problem to SAT, this is better than witing your own

Less clear that before with AI and specialized heuristics

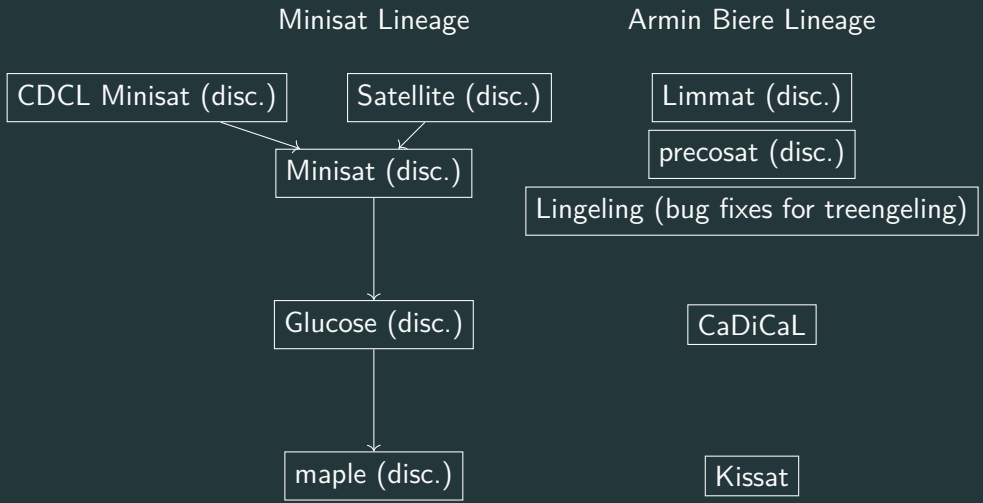
But still easier to be sure the answer is correct!



[Biere et al., POS'23], `https://cca.informatik.uni-freiburg.de/satmuseum/`


All Time Winners on SAT Competition 2025 Benchmarks
0
50
100
150
200
250
300
0
1000
2000
3000
4000
5000
Inprocessing
Preprocessing
CDCL
DPLL
319 ae−kissat−mab−2025
315 kissat−2024
307 kissat−mab−2021
302 kissat−mab−hywalk−2022
295 kissat−2020
292 sbva−cadical−2023
291 ulc−cadical−2025
275 maple−lcm−disc−cb−dl−v3−2019
266 maple−lcm−dist−cb−2018
264 maple−lcm−dist−2017
258 maple−comsps−drup−2016
222 lingeling−2014
218 lingeling−2013
217 abcdsat−2015
202 minisat−2008
197 precosat−2009
194 glucose−2012
188 glucose−2011
185 cryptominisat−2010
153 minisat−2006
150 satelite−gti−2005
133 rsat−2007
114 berkmin−2003
72 zchaff−2004
72 chaff−2001
70 limmat−2002
36 posit−1995
27 boehm1−1992

Solver Lineage



Kissat vs CaDiCaL

CaDiCaL: very controllable, much easier to understand, many features

Kissat: very fast, but more limits

Plan of Today and Tomorrow

Solving SAT fast CDCL + Inprocessing

Solving SAT right Proofs (doing the correct thing) + User propagator (doing the thing you expect)

Plan of Today and Tomorrow

What Should You Remember?

- The very basics (“we somehow resolve clauses together”)
- The motivations for things (“this explodes so we do this”)

Part I

30 Years of CDCL: SAT's Powerhorse

Introduction

SAT Problem

Input: φ in CNF

(translation from a formula to CNF is polynomial by adding new variables)

Is the formula SAT (with a model, often a counterexample or a solution) or unsatisfiable (no solution)

Approach

You can search for:

UNSAT:
- used in DP / ordered resolution / ordered paramodulation / superposition
- idea: resolve clauses until either $\bot$ was derived
- ... or no new clause is found (satisfiable)

SAT:
- Guess values...
- ... if they don't work, guess better!
- on the way find a model or accidentally unsatisfiable

Historical Note

SAT has a very tricky history:

- 1958: DP — NSA report, only available in the US
- 1961: DPLL
- $\sim$ 1995: clause learning, usually credited to [Marques-Silva and Karem A. Sakallah, ICCAD'96], but [Stallman and Sussman, AI'77]
- ... but still neither version is what we call CDCL nowadays
- 2004: SMT / DPLL(T) is created
- 2004: Lawrence Ryan creates the name CDCL, which won in the SAT community

Historical Note

New techniques can:

- come from a theory perspective store literals in the watch lists [Ryan, 04]
- come from an implementation perspective, undescribed invariants with one stored literal [Nieuwenhuis and Oliveras, 04]

Historical Note

New techniques can:

- come from a theory perspective — store literals in the watch lists [Ryan, 04]
- come from an implementation perspective, undescribed — invariants with one stored literal [Nieuwenhuis and Oliveras, 04]

- sometimes not work... — rephasing
- ...until they do later
- sometimes techniques are not useful, but have side effects that are important

Reading code is required. Some things are not described anywhere.

DPLL

DPLL Procedure

DPLL (Davis-Putnam-Logeman-Loveland) or DLL procedure [1962] tries all assignments

Procedure:

1. propagate as much as possible
2. guess a value
3. if there is a conflict, change your last guess

(the number of hands is the reason we need formal methods in LLMs)

DPLL Example: Implication Graphs

$\neg A \vee B$, $\neg B \vee C$, $\neg C \vee D$
$\neg A' \vee B'$, $\neg B' \vee C'$,
$\neg B' \vee \neg C' \vee D'$, $\neg D' \vee E'$
$\neg D \vee \neg D' \vee \neg E'$

DPLL Example: Implication Graphs

A

$\neg A \vee B$, $\neg B \vee C$, $\neg C \vee D$
$\neg A' \vee B'$, $\neg B' \vee C'$,
$\neg B' \vee \neg C' \vee D'$, $\neg D' \vee E'$
$\neg D \vee \neg D' \vee \neg E'$

DPLL Example: Implication Graphs

$A \longrightarrow B$

$\neg A \vee B$, $\neg B \vee C$, $\neg C \vee D$
$\neg A' \vee B'$, $\neg B' \vee C'$,
$\neg B' \vee \neg C' \vee D'$, $\neg D' \vee E'$
$\neg D \vee \neg D' \vee \neg E'$

DPLL Example: Implication Graphs

$A \longrightarrow B \longrightarrow C$

$\neg A \vee B$, $\neg B \vee C$, $\neg C \vee D$
$\neg A' \vee B'$, $\neg B' \vee C'$,
$\neg B' \vee \neg C' \vee D'$, $\neg D' \vee E'$
$\neg D \vee \neg D' \vee \neg E'$

DPLL Example: Implication Graphs

$A \longrightarrow B \longrightarrow C \longrightarrow D$

$\neg A \vee B$, $\neg B \vee C$, $\neg C \vee D$
$\neg A' \vee B'$, $\neg B' \vee C'$,
$\neg B' \vee \neg C' \vee D'$, $\neg D' \vee E'$
$\neg D \vee \neg D' \vee \neg E'$

DPLL Example: Implication Graphs

$A \longrightarrow B \longrightarrow C \longrightarrow D$ Level 1

$\neg A \vee B$, $\neg B \vee C$, $\neg C \vee D$
$\neg A' \vee B'$, $\neg B' \vee C'$,
$\neg B' \vee \neg C' \vee D'$, $\neg D' \vee E'$
$\neg D \vee \neg D' \vee \neg E'$

- - - - - - - - - - - - - - - - - - -

Level 2

DPLL Example: Implication Graphs

$\neg A \vee B$, $\neg B \vee C$, $\neg C \vee D$
$\neg A' \vee B'$, $\neg B' \vee C'$,
$\neg B' \vee \neg C' \vee D'$, $\neg D' \vee E'$
$\neg D \vee \neg D' \vee \neg E'$

A $\longrightarrow$ B $\longrightarrow$ C $\longrightarrow$ D Level 1

- - - - - - - - - - - - - - - - - - -

A' Level 2

DPLL Example: Implication Graphs

$\neg A \vee B$, $\neg B \vee C$, $\neg C \vee D$
$\neg A' \vee B'$, $\neg B' \vee C'$,
$\neg B' \vee \neg C' \vee D'$, $\neg D' \vee E'$
$\neg D \vee \neg D' \vee \neg E'$

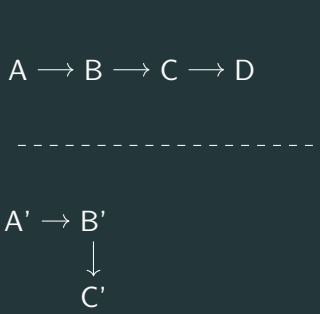
A $\longrightarrow$ B $\longrightarrow$ C $\longrightarrow$ D Level 1

- - - - - - - - - - - - - - - - - - -

A' $\longrightarrow$ B' Level 2

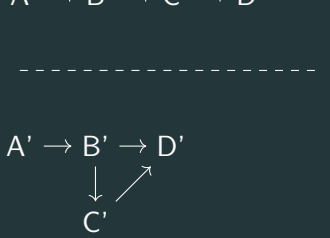
DPLL Example: Implication Graphs

$\neg A \vee B$, $\neg B \vee C$, $\neg C \vee D$
$\neg A' \vee B'$, $\neg B' \vee C'$,
$\neg B' \vee \neg C' \vee D'$, $\neg D' \vee E'$
$\neg D \vee \neg D' \vee \neg E'$

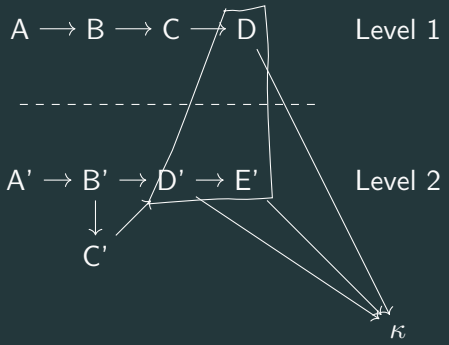


DPLL Example: Implication Graphs

$\neg A \vee B$, $\neg B \vee C$, $\neg C \vee D$
$\neg A' \vee B'$, $\neg B' \vee C'$,
$\neg B' \vee \neg C' \vee D'$, $\neg D' \vee E'$
$\neg D \vee \neg D' \vee \neg E'$

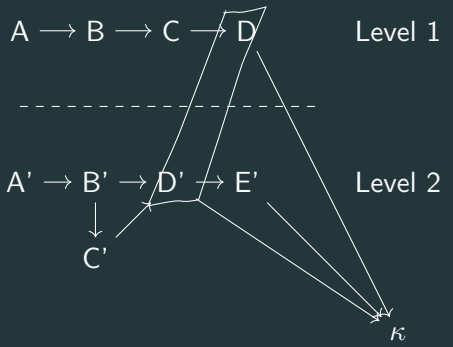


DPLL Example: Implication Graphs



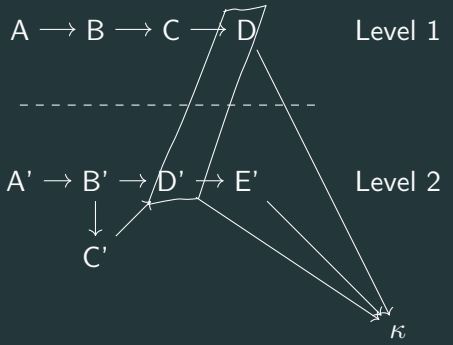
$\neg A \vee B$, $\neg B \vee C$, $\neg C \vee D$
$\neg A' \vee B'$, $\neg B' \vee C'$,
$\neg B' \vee \neg C' \vee D'$, $\neg D' \vee E'$
$\neg D \vee \neg D' \vee \neg E'$

DPLL Example: Implication Graphs



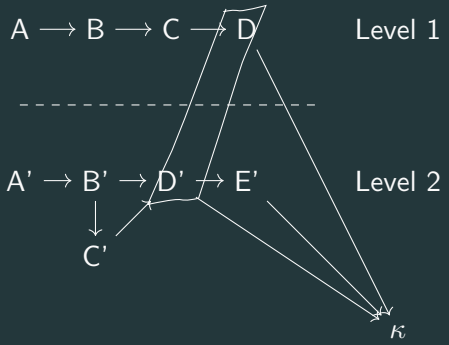
$\neg A \vee B$, $\neg B \vee C$, $\neg C \vee D$
$\neg A' \vee B'$, $\neg B' \vee C'$,
$\neg B' \vee \neg C' \vee D'$, $\neg D' \vee E'$
$\neg D \vee \neg D' \vee \neg E'$
$\neg D \vee \neg D'$ can be obtained by resolution!

DPLL Example: Implication Graphs



$\neg A \vee B$, $\neg B \vee C$, $\neg C \vee D$
$\neg A' \vee B'$, $\neg B' \vee C'$,
$\neg B' \vee \neg C' \vee D'$, $\neg D' \vee E'$
$\neg D \vee \neg D' \vee \neg E'$
$\neg D \vee \neg D'$ can be obtained by resolution!
We are going to ignore literals on level 0 (no decisions required), because they are unconditionally true

DPLL Example: Implication Graphs



$\neg A \vee B$, $\neg B \vee C$, $\neg C \vee D$
$\neg A' \vee B'$, $\neg B' \vee C'$,
$\neg B' \vee \neg C' \vee D'$, $\neg D' \vee E'$
$\neg D \vee \neg D' \vee \neg E'$
$\neg D \vee \neg D'$ can be obtained by resolution!

We are going to ignore literals on level 0 (no decisions required), because they are unconditionally true

DPLL does not need to keep track of propagations (arrows), but CDCL does

DPLL Complexity

- DPLL is exponential in the worst case...
- ... reached for pigeonhole

- actually complexity is 2^n tree exploration
- there are better bounds for k-SAT $\mathcal{O}(1.32793^n)$ as practical as matrix multiplication

- Can be trivially extended to count models (#SAT) or to find cheapest model (MaxSAT)

DPLL Implementation

- Often recursive back then (does not work anymore) also many assumptions on input
- Based on counters (count the number of false literals per clause)
- Targeted finding SAT MOMS decision heuristic

Clause-Learning

CDCL Procedure

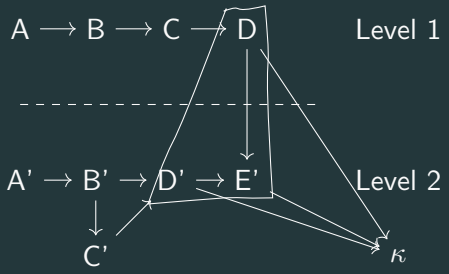
Idea: Using DPLL to guide resolution.

Instead of resolving randomly, you resolve along arrows!

Until you found a model or derive an empty clause:

1. propagate
2. if conflict, analyze and backjump or return unsatisfiable (see next slide)
3. if all variables set, return satisfiable
4. otherwise, decide a variable

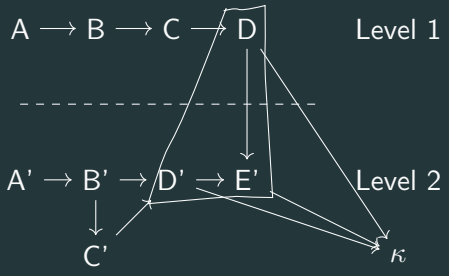
CDCL Conflict Analysis



Aim: find a cut such that every path from a decision to κ is covered. Pick one where there is only one literal on highest level. Every step of going

back = one resolution step with the reason of the propagation.

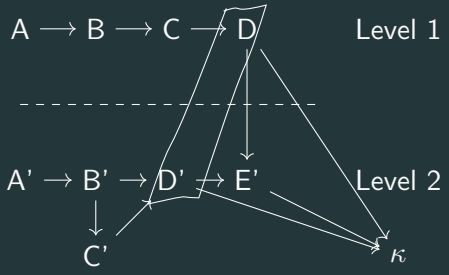
CDCL Conflict Analysis



You can ignore all paths (and all decisions) that don't lead to κ (bad decision heuristic!) After

finding a cut, learn the clause and propagate it by *backjumping* to the right level.

CDCL Conflict Analysis



You can ignore all paths (and all decisions) that don't lead to κ (bad decision heuristic!) After

finding a cut, learn the clause and propagate it by *backjumping* to the right level.

CDCL Conflict Analysis

$A \longrightarrow B \longrightarrow C \longrightarrow D \rightarrow \neg D'$ Level 1

- - - - - - - - - - - - - - - - - - -

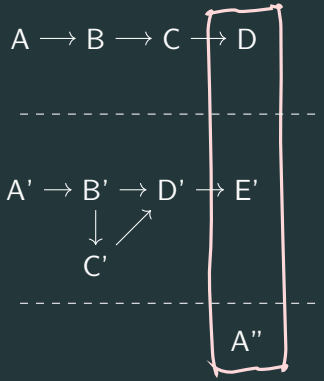
You can ignore all paths (and all decisions) that don't lead to κ (bad decision heuristic!) After finding a cut, learn the clause and propagate it by *backjumping* to the right level.

Intermezzo: Why the first UIP?

Question

Should you backtrack early or late?

Intermezzo: Why the first UIP?



Assuming learned clause is $\neg D \vee \neg E' \vee A''$, literals can you set for the clause to be useful? or is the last UIP $\neg A \vee \neg A' \vee A''$ better?

Conflict-Clause Minimization

If you can remove literals without adding new ones, do it [Sörensson and Biere, SAT'09]

Removes around 30% of all literals.

Quadratic in the size of implication graph, linear if you remove also the failing attempts [Van Gelder, SAT'09].

Conflict-Clause Minimization

If you can remove literals without adding new ones, do it [Sörensson and Biere, SAT'09]

Removes around 30% of all literals.

Quadratic in the size of implication graph, linear if you remove also the failing attempts [Van Gelder, SAT'09].

There are also some attempts at reducing the size without making the clause worse [Fleury and Biere, SAT'21]

CDCL Properties

Theorem (Proof of False)

If you find a conflict at level 0, conflict analysis will derive $\bot$.

Theorem (Proof of False)

If you find a conflict at level 0, conflict analysis will derive $\bot$.

Theorem (Termination)

CDCL terminates

Proof.

Take the trail $M_0 ... M_n$, and $\mu : L \mapsto \begin{cases} 0 & \text{if propagation} \\ 1 & \text{if decision} \end{cases}$. Then $\mu(M_0)...\mu(M_n) \cdot \underbrace{2...2}_{\#\text{unset lits}}$ is lexicographically decreasing. □

Gives $\mathcal{O}(3^n)$, but can be easily tightened to $\mathcal{O}(2^n)$.

CDCL Properties

Theorem (Termination)

CDCL does not relearn a clause

Proof.

We eager propagate, so we would have propagated. □

Theorem (Termination)

CDCL does not learn (ordered-) redundant clauses

Proof.

[Weidenbach, 15] □

CDCL Complexity

- CDCL is exponential in the worst case...
- ... reached for pigeonhole... practically PH often faster for DPLL, because learning useless
- ... but also exponentially better than DPLL

- if you keep all clauses, your algorithm becomes EXPSPACE
- With new variables, proofs can be exponentially shorter

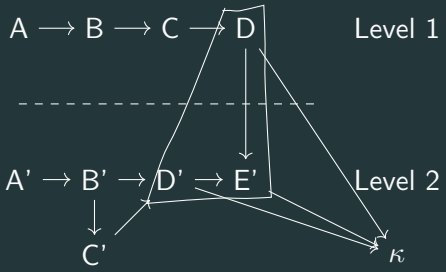
Decision Heuristics and Phase Saving

Old Approach Try to satisfy as many clauses as possible enable pure literal detection

Decision Heuristics and Phase Saving

Old Approach Try to satisfy as many clauses as possible enable pure literal detection

New Approach Focus on variables involved in recent conflicts.

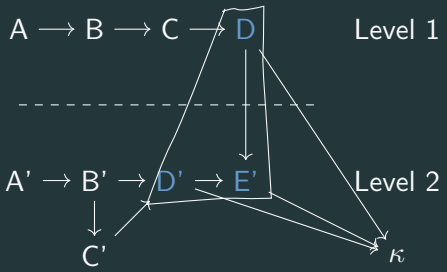


Decision Heuristics and Phase Saving

Old Approach Try to satisfy as many clauses as possible — enable pure literal detection

New Approach Focus on variables involved in recent conflicts.

1. very old: bump only conflicts — close to the old version

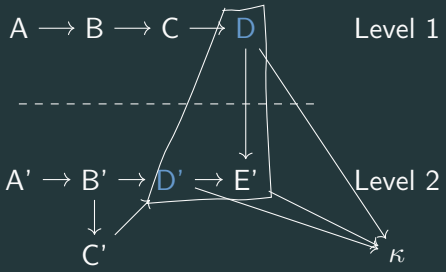


Decision Heuristics and Phase Saving

Old Approach Try to satisfy as many clauses as possible — enable pure literal detection

New Approach Focus on variables involved in recent conflicts.

1. very old: bump only conflicts — close to the old version
2. old: bump the literals during resolutions (but not the conflict)

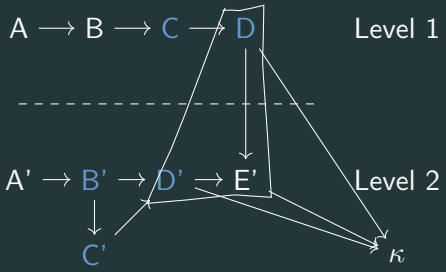


Decision Heuristics and Phase Saving

Old Approach Try to satisfy as many clauses as possible — enable pure literal detection

New Approach Focus on variables involved in recent conflicts.

1. very old: bump only conflicts — close to the old version
2. old: bump the literals during resolutions (but not the conflict)
3. new: bump the literals during resolutions and their reason [Liang et al., SAT'16]



Decision Heuristics and Phase Saving

Old Approach Try to satisfy as many clauses as possible — enable pure literal detection

New Approach Focus on variables involved in recent conflicts.

1. very old: bump only conflicts — close to the old version
2. old: bump the literals during resolutions (but not the conflict)
3. new: bump the literals during resolutions and their reason [Liang et al., SAT'16]

Phases Remember how things were set last time and do the same

And in Reality?

```
$ cadical-3.0.0 eq.atree.braun.9.unsat.cnf --stats --plain
c  seconds  reduction conflicts       size tier2      variables
c         MB    restarts   redundant    glue  trail        remaining
c          level     rate       size/glu tier1  irredundant
c
c {  0.00  5  0 0   0  0    0    0 0.0 0 0 2 6  0% 3003 889 100%
c -  0.02  6 13 1   0  1  301  133 6.2 63 10 2 6 100% 2969 889 100%
c -  0.03  6 13 2   1  1  735  274 5.4 52 10 2 6 95% 2969 889 100%
c }  0.04  6 14 2   3  1 1005  527 4.8 45 9 2 6 90% 2969 889 100%
...
c conflicts:                289269     32777.91    per second
```

Which Clauses to Keep?

- Old idea: keep all — propagation roughly quadratic in practice
- keep short clauses
- LBD or glue: use the number of levels in the implication graph [Simon and Audemard, 2009]
- ... and update it — actually different definitions, off by 1

Which Clauses to Keep?

Deletion in geometric number of conflicts (Glucose), instead of geometric intervals (MiniSAT)

[Gstrein et al, SAT'25]

1. keep used clauses
2. keep all very short clauses $\mathrm{size} <= 2$
3. give *used* short clauses a second chance $\mathrm{size} <= 6$
4. remove 75% of all clauses

(size or LBD does not make a huge difference)

Glucose uses glue instead of size

idea from cache eviction

you need both criteria at least

Which Clauses to Keep?

Dynamic limits looks nicer (we have LBD <= 30), but not helping practically:

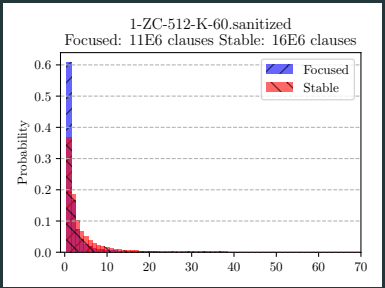


Figure 1: LBD distribution for one problem from the SAT Competition

Which Clauses to Keep?

Dynamic limits looks nicer (we have LBD <= 30), but not helping practically:

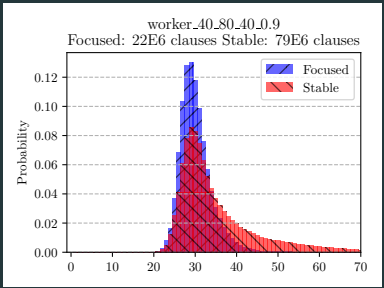


Figure 1: LBD distribution for one problem from the SAT Competition

Which Clauses to Keep?

Dynamic limits looks nicer (we have LBD <= 30), but not helping practically:

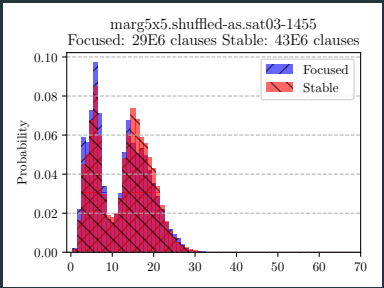


Figure 1: LBD distribution for one problem from the SAT Competition

Which Clauses to Keep?

Dynamic limits looks nicer (we have LBD <= 30), but not helping practically:

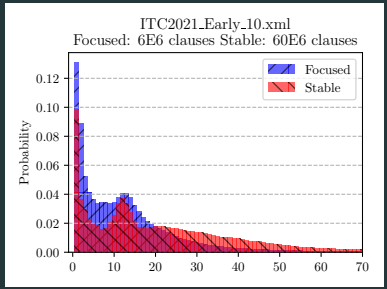


Figure 1: LBD distribution for one problem from the SAT Competition

And Restarts?

Theoretical result: we need restarts for expressiveness for UNSAT at least we think we need

Restarts are helpful to learn short clauses and find short proofs

Restarts shuffle decisions, making it possible to learn similar clauses

And Restarts?

Theoretical result: we need restarts for expressiveness for UNSAT at least we think we need

Restarts are helpful to learn short clauses and find short proofs

Restarts shuffle decisions, making it possible to learn similar clauses

But not useful for satisfiable

contrary to old intuition

And Restarts?

Idea: if the clauses you are currently learning look bad, undo and restart! [Biere and Fröhlich, POS'18]

Counteract too many restarts [Ramos et al, SAT'11] decision reuse, needs phase saving

Portfolio approach: alternate between many restarts and few restarts [Chanseok Oh, SAT'15]

Portfolio approach: alternate between decision heuristics (stable vs aggressive)

Rephasing: sometimes overwrite your saved phases target phases, best phases, OIF#C

Watched Literals

Status of a clause can be summarized to two literals:

- if one is true, the other has higher level, so does not matter
- both false, conflict
- one false, one unset: awaits update
- two unset: does not matter

Standard implementation uses a *third* literal inlined in the watch lists

When propagating L, visit only the clauses watching $-L$. always update blocking literal?

You need to sort learnt clause.

Watched Literals Propagation (without Blocking Literals)

```
1:  for clause C in watch list WL of false literal L (C watching L) do
2:      Let L' be the other watched
3:      if no non-false literal in unwatched literals of C then
4:          if L' is false then
5:              C is a conflict
6:          else
7:              propagate L' due to C.
8:      else
9:          let K be a non-false unwatched literal of C
10:         remove C from the WL    ▷ implicit: modify WL in place and don't keep it!
11:         add C to the watch list of K
```

The Important Ingredients to make it work

Efficient propagation (I) Watch literals instead of counters [Moskewicz et al, DAC'01]

Efficient propagation (II) blocking literals — good for CPUs!

Aggressively remove clauses don't try to simplify them

Fast decision heuristics fast matters more than quality

Lots of engineering no one-approach-fits-all — mostly in next part

Part II

15 Years of Inprocessing

Introduction

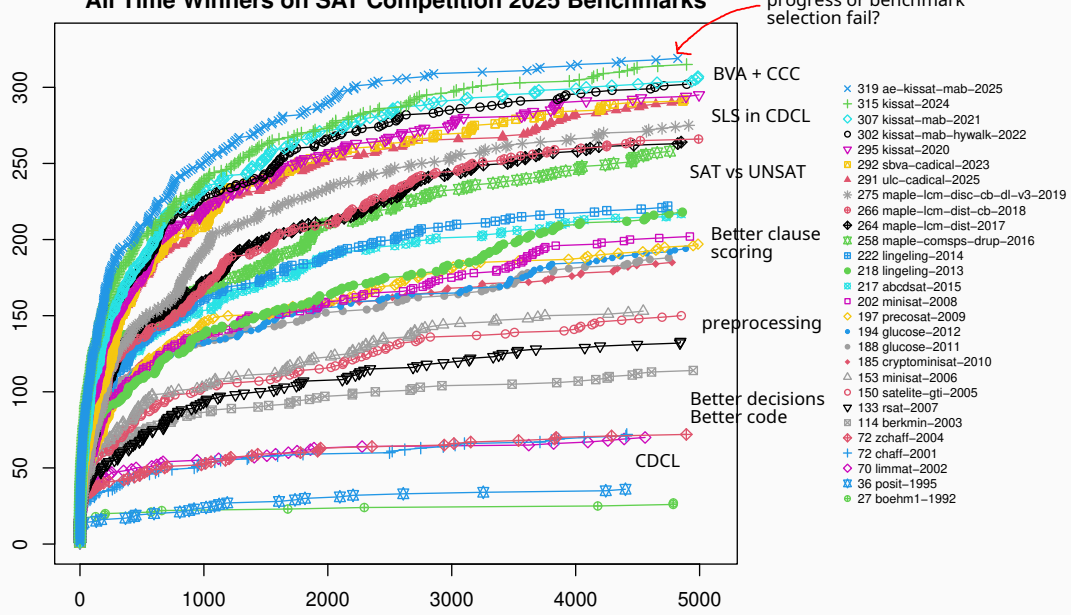
Introduction

CDCL is good at some things, but not at others

So: combine CDCL with other techniques.

In essence, any technique you can think of can be used.

Lots of work to find useful things that work.


All Time Winners on SAT Competition 2025 Benchmarks
progress or benchmark selection fail?
BVA + CCC
SLS in CDCL
SAT vs UNSAT
Better clause scoring
preprocessing
Better decisions
Better code
CDCL
319 ae–kissat–mab–2025
315 kissat–2024
307 kissat–mab–2021
302 kissat–mab–hywalk–2022
295 kissat–2020
292 sbva–cadical–2023
291 ulc–cadical–2025
275 maple–lcm–disc–cb–dl–v3–2019
266 maple–lcm–dist–cb–2018
264 maple–lcm–dist–2017
258 maple–comsps–drup–2016
222 lingeling–2014
218 lingeling–2013
217 abcdsat–2015
202 minisat–2008
197 precosat–2009
194 glucose–2012
188 glucose–2011
185 cryptominisat–2010
153 minisat–2006
150 satelite–gti–2005
133 rsat–2007
114 berkmin–2003
72 zchaff–2004
72 chaff–2001
70 limmat–2002
36 posit–1995
27 boehm1–1992
0
50
100
150
200
250
300
0
1000
2000
3000
4000
5000

Local Search

Local Search

CDCL = complete approach

SLS (stochastic local search) = incomplete approach

Idea: guess a model and then flip literals one at a time!

Incomplete Search

Local search algorithm:

```
A, a total model
while (A is not a model) {
  pick one literal of A
  flip it: A := A - {L} + {-L}
}
```

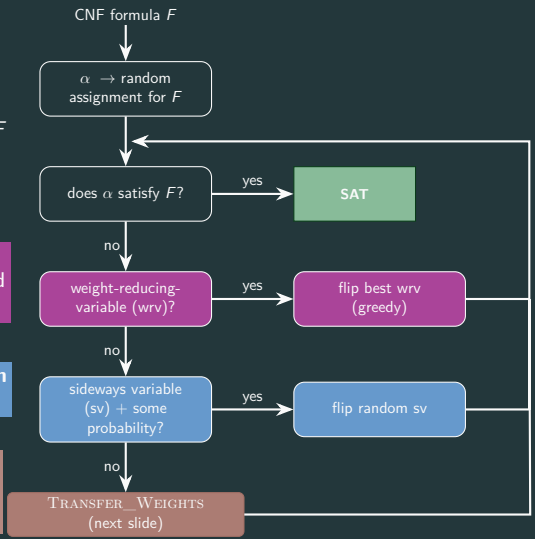
Why?

- works best for satisfiable random instances
- works best for some hard combinatorial benchmarks
- you never know if you instance does not end up with two independent components (one trivially SAT, one hard unsat) No evidence that it is happening though
- weakness of SLS is propagation, which is strength of CDCL

The DDFW Algorithm (TaSSAT) [Chowdhury et al., NFM'23, TACAS'24]

Input: CNF formula F, w_0, spt, $cspt$, $c_>$, $c_\leq$
Output: Satisfiability of F

1: $W(C) \leftarrow w_0$ for all $C \in F$
2: $\alpha \leftarrow$ random truth assignment on the variables in F
3: **for** $f = 1$ **to** MAXFLIPS **do**
4: **if** α satisfies F **then**
5: **return** SAT
6: **else**
7: **if** there is a wrv **then**
8: Flip a wrv that most reduces the falsified weight
9: **else if** there is a sv and some probability **then**
10: Flip a sideways variable
11: **else**
12: TRANSFER_WEIGHTS(F, w_0, spt, $cspt$, $c_>$, $c_\leq$)
13: **return** "No model found"



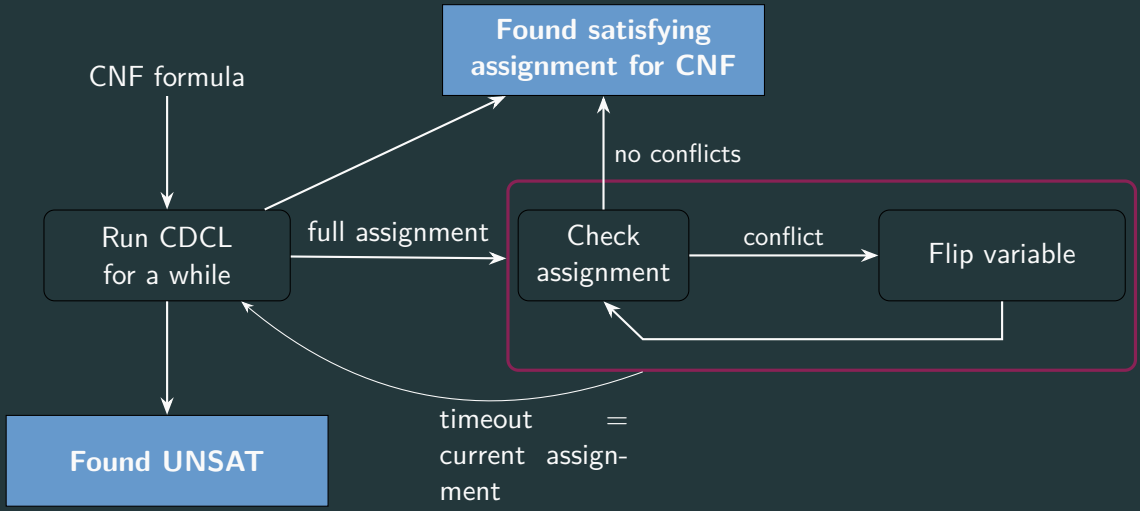
Other SLS Algorithms

ProbSAT: randomly select a clause, randomly (weighted) select a literal of that clause to flip

CCAnr: use *configuration*, only flip if a neighboring literal was flipped recently

Local Search and CDCL

Local Search and CDCL



Inprocessing: do what CDCL is bad at

Overview

Equivalent Literal Substitution (ELS)

Binary Implication Graph (BIG): draw implications from the binary clauses

If there are cycles, the literals of the cycles are all equivalent. So substitute them in the problem.

Example: $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$

Equivalent Literal Substitution (ELS)

Binary Implication Graph (BIG): draw implications from the binary clauses

If there are cycles, the literals of the cycles are all equivalent. So substitute them in the problem.

Example: $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$

Useful because:

- makes the data structures more compact
- removes clauses, shorten causes
- helps heuristics (score is not split across both variables)

Bounded Variable Elimination

Original DP, but with bounds

most important preprocessing technique

$A \vee B \qquad \neg A \vee \neg B \quad \phi$ without A, $\neg$A
$A \vee C \vee D \quad \neg A \vee D$
resolve all clauses together on A: $B \vee D, B \vee D, \neg B \vee C \vee D, \phi$

Do it only if it reduces the number of clauses

or does not add too many

Bounded Variable Elimination

Original DP, but with bounds

most important preprocessing technique

$A \vee B \quad \neg A \vee \neg B \quad \phi$ without A, $\neg$A
$A \vee C \vee D \quad \neg A \vee D$
resolve all clauses together on A: $B \vee D, B \vee D, \neg B \vee C \vee D, \phi$

Do it only if it reduces the number of clauses

or does not add too many

Requires model reconstruction by setting A (if all clauses containing $\neg A$ are satisfied) or $\neg A$ (otherwise).

Bounded Variable Addition

Opposite of BVE, but only if you decrease the number of clauses.

Initially only used to fixed bad at-most-one encoding coming from biology

Come-back as structured BVA in 2023, although structured not important better implementation in [Przybocki et al., 2026]

Basically creates AND gates [Przybocki et al., SAT'26], works for XORs too [Pollitt et al., SAT'26] unclear though if useful beyond pigeonhole

Intermezzo: Strengthening Clauses

Question

Assume a clause is used to strengthen a clause, should you mark it as used for CDCL?

Strengthening Clauses

Most used techniques: backward subsumption and... Bloom filter hashing
For each new clause:

1. traverse all clauses sharing one literal
2. subsume or strengthen

... forward subsumption single-watched scheme, but not complete (multiple rounds)

... and vivification = forward subsumption with multiple resolution steps at once, reusing propagation [Somenzi and Alembic, DAC'07], [Luo et al., IJCAI'17], [Pollitt et al, POS'25]

Can remove subsumed clauses or remove redundant literals or remove irredundant clauses

Run in increasing intervals, scheduled on number of conflicts, limited in time

And More?

General framework [Järvisalo et al., IJCAR'12]

Clausal Congruence Closure: extract gates out of the CNF and merge output of identical gates [Biere et al, SAT'24]

Sweeping: use a small SAT solver to find equivalences of literals [Biere et al, FMCAD'24] not useful since 2021

Gates: BVE on gate output needs fewer resolvents (only with gate clauses, not all) [Eén and Biere, SAT'05]

Binary transitive reduction: remove implied binary clauses

Ternary resolution: resolve all ternary clauses that produces binary or ternary clauses

And More?

Optional ones:

Autarkies: removes part of the of the clauses if you find a partial assignment that makes no clause false — not harmful though

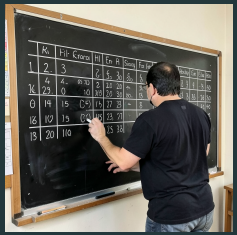
BCE: Blocked clause elimination: remove some clauses. Not useful since around 2015

BCA: Blocked clause addition: add blocked clause. Not efficient enough, except when fine-tune for some families

Instantiation: removes some literals from clauses, too expansive unless used during vivification (last literal only)

Gaussian Elimination: native XOR reasoning (with recovering XOR clauses)

What is Hard?



Good Implementation the code must be fast enough

Scheduling run often enough to be useful

Never Slow the code should not have too bad behavior — limiting the length

Never Too Bad the code should not cost you too much if useless — skipping iterations

Scheduling

Some techniques can be run until fix point, but it is rare. Most techniques need too much time.

Usually: 60% / 30% / 10%: for 60% no limit is enough; for 30% you need to limit the technique otherwise the technique will run for too long; for 10%, something weird happens but you cannot easily fix it.

hard to find the 30%

Scheduling

Some techniques can be run until fix point, but it is rare. Most techniques need too much time.

Usually: 60% / 30% / 10%: for 60% no limit is enough; for 30% you need to limit the technique otherwise the technique will run for too long; for 10%, something weird happens but you cannot easily fix it. hard to find the 30%

Current state-of-the-art (in CaDiCaL and Kissat):

Theorem (ticks)

Estimation of the number of cache misses.

In most parts of the SAT solver, easy (each clause access is one cache miss), although does not fully work in other parts (like local search)

Redundant vs Irredundant

Initially: all clauses N are *irredundant* (equivalent to initial set of clauses).

Some irredundant clause can be deleted $A \vee B \vee C$ irredundant with $B = \neg C$

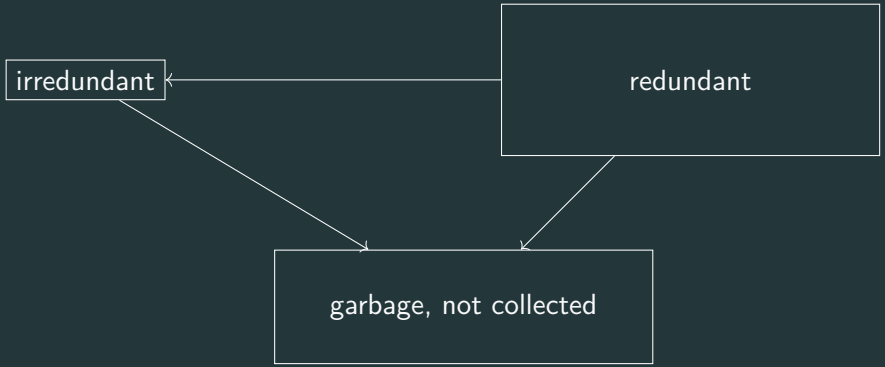
Some redundant clause U can be promoted to be irredundant $A \vee B \vee C$ irred, $A \vee B$ redundant

At every point, irredundant are equisatisfiable to the initial set of clauses. All redundant clauses can be removed

Usually $N \models U$, but is not necessary. after BVE, keep some clauses in U to be deleted soon

Redundant vs Irredundant

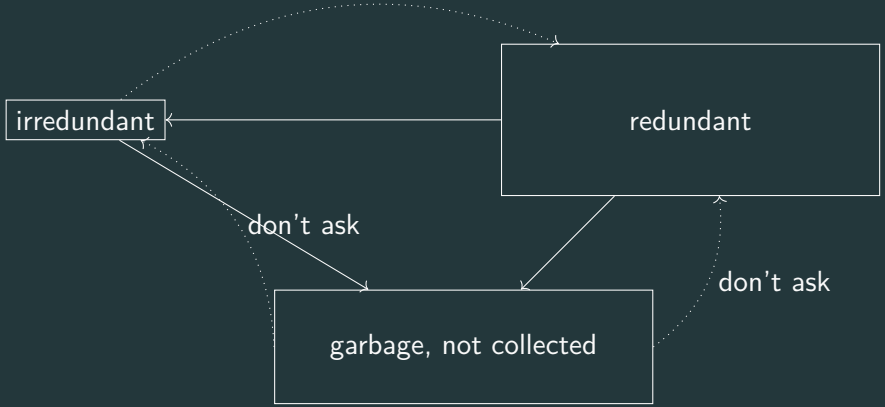
Clauses in a solver:



Goal of inprocessing: simplify irredundant (shorter clauses, fewer clauses) old approach
but also for redundant clauses that you have a chance to keep modern approach

Redundant vs Irredundant

Clauses in a solver:



Goal of inprocessing: simplify irredundant (shorter clauses, fewer clauses) old approach
but also for redundant clauses that you have a chance to keep modern approach

Today's Exercise: Probing

Idea: let's try to find as many unit and binary clauses as possible (helps for ELS!)

- Every propagation on level 1 can be done by a binary clause
- Every conflict on level 1 is a unit

Probing

```
for (all literals lit) {
  decide (lit);
  propagate_binaries (lit);
  for (all still propagating clauses) {
     either strengthen the clause to a binary clause
     or
     learn a new binary clause using the first UIP
  }
  if (conflict)
    learn (-lit)
  backtrack (0)
}
```

CaDiCaL

```
for (auto lit : lits) {
  if (!active (lit))
    continue;
  // now do something
}

Clause *c;
mark_garbage (c); // to delete a clause
```

Conclusion

Conclusion

CDCL is the main part of the SAT solver...

... but you need the rest to solve more benchmarks

A lot of work in SAT solvers is getting the implementation *right* and *fast*

Part III

20 Years of Proofs

Welcome Back

Today: getting stuff right!

- no bugs
- controlling the search w.r.t. your knowledge

Some Motivation

Yesterday, we talked about how SAT solvers work with technical details.

Getting the SAT solver correct is extremely challenging, due to the many techniques.

Interactions between techniques can become messy. only testing not enough, but necessary

Fuzzing and Testing

Generate instances and check the reasoning.

Some (De)Motivation

Yesterday, we talked about how SAT solvers work with technical details.

But: focus on hard instances and progress in SAT solvers does *not* correlate with progress in *applications*. MaxSAT's win from ILP, not SAT

Not all time limits are made equal lucky search

Not all techniques are made equal IsaSAT winning the EDA challenge

But: if you throw garbage at it, SAT solvers are pretty good at handling it.

Some (De)Motivation

Yesterday, we talked about how SAT solvers work with technical details.

Sometimes, SAT also the wrong place to do things symmetries lost after encoding, not stable

IPASIR-UP

So: can you take control of the SAT solvers?

Introduction

The Problem

SAT solvers are big pieces of code: CaDiCaL is 62k lines of code.

SMT Competition, model counting have disagreements: how to solve them?

How do we get this working correctly?

printing models is simple enough

Mandatory during SAT Competition!

How to call a SAT Solver?

One-Off: call the SAT solver and be happy with the answer

Incrementally: call the SAT solver, get an answer, add clauses and ask another question

see model counting

Fine-Grained: see next part

Figure 2: Initially, you set up the clauses

Figure 3: Then you solve it, without influence

Figure 4: Then you can add clauses

Figure 5: Then you solve it, without influence

IPASIR

The standardized C interface for incremental SAT solvers. Requires solvers to implement

add() for adding clauses.

assume() to set the value of a variable for the next solve call.

solve() to determine whether the current formula under the current assumptions is satisfiable.

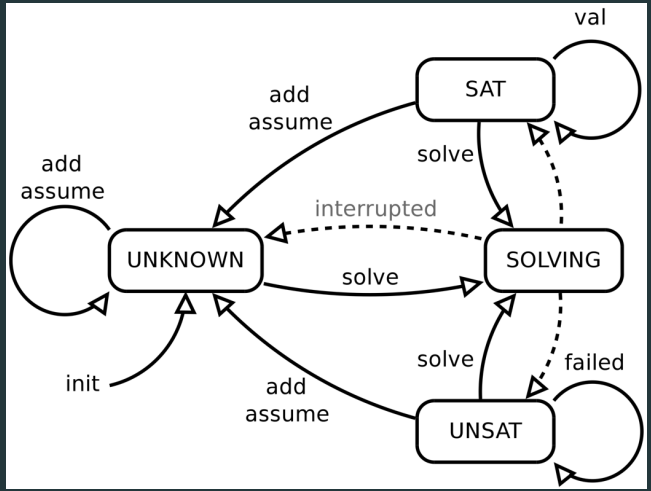
failed() to query which assumptions lead to an unsatisfiable result.

val() for retrieving the value of a variable in the model. tricky!

See also unreleased IPASIR-2. [unchanged for >2 years]

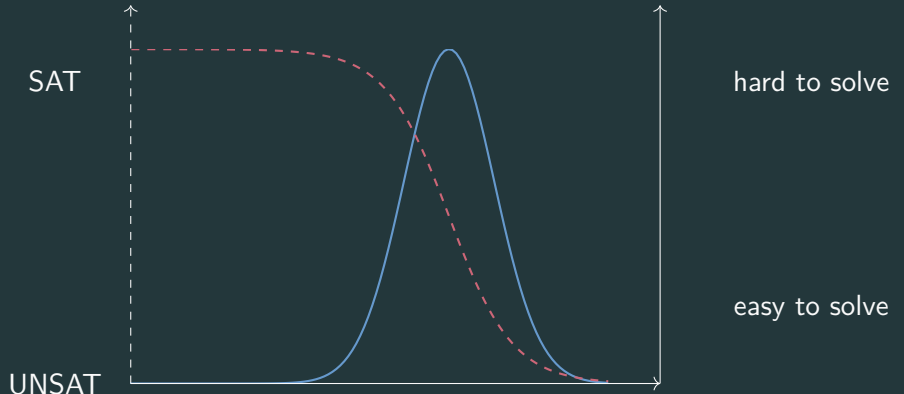
IPASIR (2)

IPASIR also defines a state machine that SAT solvers need to adhere to



Fuzzing

Random Instances



Random Instances

Important property:

- no interesting instance is random — issue in the SAT comp.
- random instances trigger everything in a SAT solver, but it can take time
- generating interesting instances critical until a few years back
- the optimal ratio comes from a practical point-of-view
- improvement with option fuzzing

Fuzzing

Generate one instance, compare outputs from multiple solvers or multiples configurations.

Outcomes:

Crashs: you need to fix them

SAT: check model

UNSAT: nothing to do without symmetry breaking, if you have a model, you can check that every learnt clause is satisfied

Fuzzing

Generate one instance, compare outputs from multiple solvers or multiples configurations.

Outcomes:

Crashs: you need to fix them

SAT: check model

UNSAT: nothing to do without symmetry breaking, if you have a model, you can check that every learnt clause is satisfied

API fuzzing better than file fuzzing

Fuzzing

Who uses it?

compilers

GUIs (firefox)

cvc5 Murxla, but also grammar-based approaches and merging instances

MaxSAT [Paxian and Biere, JAIR'26]

Nowadays bugs in SAT solvers are very rare, much less rare in SMT solvers, many MaxSAT solvers don't pass the official regression suite at the competition.

Fuzzing: Does it work?

Usually very strong and extremely fast.

When doing coverage of clausal congruence closure, I ended up fuzzing for *several weeks* of non-debug binary on a 64-core server.

One branch was reached exactly *once*.

One branch remains that I am convinced is reachable, but was not reached.

Proofs

Why Proofs?

But: what to do when the result is wrong? Or you are the only solver finding a solution?

Why Proofs?

But: what to do when the result is wrong? Or you are the only solver finding a solution?

Idea: print every learnt clause and check it independently.

Why Proofs?

But: what to do when the result is wrong? Or you are the only solver finding a solution?

Idea: print every learnt clause and check it independently.

How to check $F \models C$? Well $F, \neg C \models \bot$ is too expansive (SAT call).

SAT Redundancy Check

Practical tests:

DRUP [Heule et al.,IEEE'13] sufficient for CDCL

$$F, \neg C \vdash_1 \bot$$

where $\vdash_1$ means only unit propagation.

DRAT [Heule et al.,SAT'14] DRUP + blocked clause addition

LRAT give the exact propagation to do [Cruz-Filipe et al, CADE26]

DSR [Codel et al., FMCAD'24] Also symmetry breaking but we cannot produce those proofs

veriPB technically PB format for optimization, but also works for SAT [Gocht et al., IJCAI '20]

Practical SAT Proof Checking

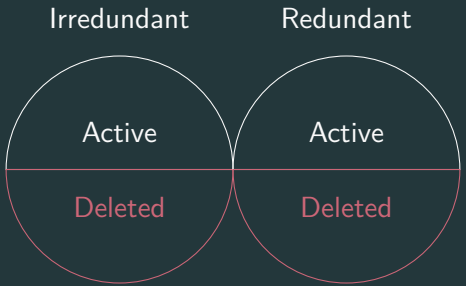
'd' deletion important for memory usage

Except for LRAT, overhead up to 10 times.

Proof trimming (removing all the useless steps) is very useful.

We have verified proof checkers.

Clauses in SAT Solvers

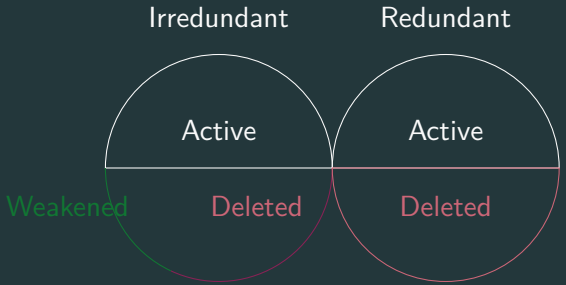


Clauses can move from redundant to irredundant while strengthening

At every point, you can remove all redundant clauses

Not all proof formats have the distinction redundant / irredundant (or core)

Incremental SAT



You have to restore = "undelete" clauses
[Fazekas et al., 2019]

Each remove clause has a witness ω. If a literal is tainted, must be restored. Also give a way to reconstruct the model.

(L)IDRUP(E) format: only check resolutions and check that model is assignment of initial set of clauses, requires model reconstruction. restoring must not be checked

Incremental SAT Redundancy Check

Practical tests:

(L)IDRUP [Fazekas et al., LPAR'24] sufficient for CDCL

$$F, \neg C \vdash_1 \bot$$

where $\vdash_1$ means only unit propagation. LRAT not possible!

LIDRUPE LIDRUP + definitions for BVA [Fazekas et al., POS'25]

eDRAT DRAT + external clauses inlined is $\bot$ a valid external addition?

And beyond SAT?

SAT is at the sweep spot, only one few (two different) rules for clause addition.
cvc5 has several hundred.

Also clauses are not too long to write.

The initial patch for proofs was $<$100 lines, and just worked.

For MaxSAT: every technique needs translation.

Proofs: Understanding

Why?

If the best proof is exponential, CDCL cannot find a better one!

DPLL = tree resolution

(CDCL in-between)

CDCL + restarts = general resolution

(above: there are exponential proofs)

CDCL + restarts + general BVA = (extended) Frege

Other Results

Now, complexity results is only interested in unsat, not satisfiable.

Very few reasonable results: VSIDS decision heuristic can give bad results

BVE is bad, because it undoes extended Frege this is a real issue on some benchmarks

Symmetry Breaking

Symmetry Breaking

Idea: If you can permute values of x and y, add the condition $x >= y$

It is now possible to generate proofs veriPB

Beware, encoding adds asymmetry (like $[abc] + [def]$, so UP better (see later)

Currently no proof that it is useful beyond encoding [Anders et al, SAT'26]

Parallel Solving

What Does Parallel Mean?

Cooperative: several solvers work on the same instances

Splitting: the problem is split into several independent instances (Cube-and-conquer, CnC)

look-ahead in March

Automatic CnC failed, but used for splitting by hand

Heule (pure SAT), Ganesh (SAT + Algebra)

Intuition

Figure 6: Zoombies (SAT solvers) attacking a castle. One of them can find the shortcut. As in real life, some are running in the wrong direction. (Mistral AI image)

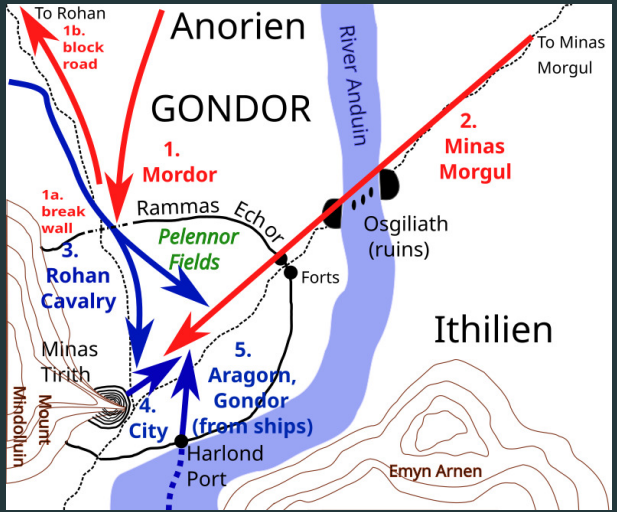


Figure 7: Old intuition: work on separate part of the search space (I run out of tokens)

Intuition

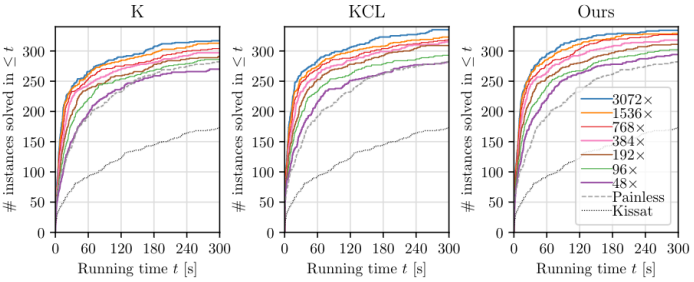


Figure 8 Strong scaling of MallobSat's baseline K and KCL configurations (left, center) and our new setup (right) in terms of absolute performance from 1 to 64 nodes (48–3072 cores). We also show running times of sequential Kissat and 48-core PL-PRS-BVA-KISSAT ("Painless").

Figure 8: Mallob in different configurations (K = Kissat only, KCL = Kissat + CaDiCaL + Lingeling, Ours = less diversification)

Intuition

Strong consequence:

- sharing many clauses is not very important... gimsatul build to share clauses, but mostly useless
- sharing few good clauses is important mallob
- varying the strategy not really important
- for less than 64 threads, proofs seem to be constant size[Fleury and Biere, POS'22]

Part IV

3 Years of User Propagator

Example

How can we use this in our graph coloring encoding where $e_{u,v}$ is true whenever u and v have the same color?

Problem: The encoding requires $O(n^3)$ clauses that might not be necessary. This does not scale to larger graphs.

Example

How can we use this in our graph coloring encoding where $e_{u,v}$ is true whenever u and v have the same color?

Problem: The encoding requires $O(n^3)$ clauses that might not be necessary. This does not scale to larger graphs.

1. Start with some or no clauses (e.g., constrain the number of colors).
2. Obtain a model for this underconstrained version.
3. Check if the model satisfies all (also unadded) clauses.
4. Any constraints not satisfied: add (some of) them and go to 2.
5. Otherwise, we have found a solution.

Example

How can we use this in our graph coloring encoding where $e_{u,v}$ is true whenever u and v have the same color?

Problem: The encoding requires $O(n^3)$ clauses that might not be necessary. This does not scale to larger graphs.

1. Start with some or no clauses (e.g., constrain the number of colors).
2. Obtain a model for this underconstrained version.
3. Check if the model satisfies all (also unadded) clauses.
4. Any constraints not satisfied: add (some of) them and go to 2.
5. Otherwise, we have found a solution.

Will this always find a solution?

Limitations

Problem: Incremental solving does not allow interactions with the solver during solving.

Limitations

Problem: Incremental solving does not allow interactions with the solver during solving.

- Solver always runs to completion, even when violating (unadded) clauses early.
- We cannot prevent solver from "taking a wrong turn".
- Requires domain knowledge to be concisely representable in propositional logic.
- Hard to judge from the model which unsatisfied clauses are important.

Figure 9: The user is the labrador

User Propagators

Definition

(User) Propagators are a general concept that refers to methods that propagate knowledge to the solver.

They have been around in many declarative solvers (e.g., SMT, CP, MIP, even the clasp SAT solver).

In the SAT context, this was usually done by hacking the SAT solver

Problem: Requires you to use one version of one SAT solver (no easy updates).

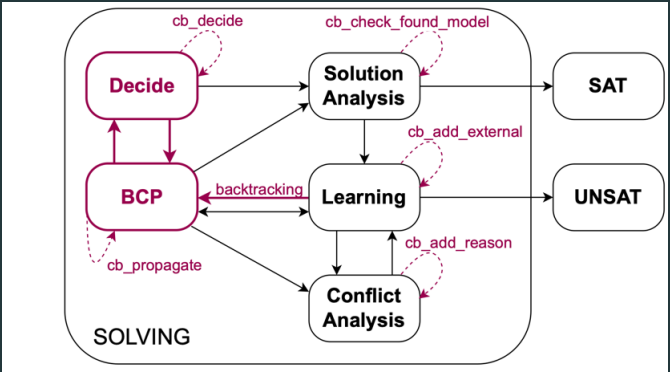
Problem: Requires intimate knowledge of the SAT solver code.

IPASIR-UP

IPASIR-UP [Fazekas et al, JAIR'24] is the extension of IPASIR to User Propagators (UP) that allows a solver agnostic implementation of user propagators in the SOLVING state.

IPASIR-UP

IPASIR-UP [Fazekas et al, JAIR'24] is the extension of IPASIR to User Propagators (UP) that allows a solver agnostic implementation of user propagators in the SOLVING state.



IPASIR-UP (2)

Propagators and observed variables are registered with the solver. IPASIR-UP calls the appropriate propagator functions during the solving process.

IPASIR-UP defines two different types of functions:

Notifications inform the propagator of state changes.

Callbacks ask the propagator for input.

Basic Callbacks

Decide allows the propagator to decide the next variable assignment.

Add Clause allows adding a new clause.

Check Model allows declaring that a model is not really a model. This requires either a new variable or a new clause.

Basic Notifications

New Assignment informs that a new variable assignment (decision or propagation) has been added.

New Level indicates a new decision level.

Backtrack indicates that the solver backtracked to a past decision level. This is a notification that implicitly informs the propagator that variables have been unassigned.

Propagations

Propagating a literal is easily done via callback.

The tricky part is that every propagation needs a reason clause. Why?

Propagations

Propagating a literal is easily done via callback.

The tricky part is that every propagation needs a reason clause. Why?

This reason clause can be added eagerly via callback, otherwise, the solver will ask for it during conflict analysis.

Reason clauses must be propagating.

Learned Clauses

IPASIR-UP does not provide any information about learned clauses.

However, the IPASIR interface allows registering a callback that obtains all learned clauses.

User Propagator Example

Graph Coloring

We want to lazily generate the clauses in our graph coloring encoding, where a variable $c_{u,\ell}$ indicates whether vertex u has color ℓ out of k possible colors.

We start by adding no clauses and letting the solver guess colors arbitrarily.

How would you implement a propagator?

State

How does our state look like? What do we need to track?

State

How does our state look like? What do we need to track?

- How many colors have been excluded.
- Whenever a variable $c_{u,\ell}$ is set to false, we increment the counter for the vertex.
- In case of backtracking, we decrement the counters and unmark the vertices.

Propagations (1)

Whenever the exclusion counter for any vertex reaches $k-1$ we propagate $c_{u,i}$, where i is the one color not yet unassigned.

What is the reason clause in case of a conflict?

Propagations (1)

Whenever the exclusion counter for any vertex reaches $k-1$ we propagate $c_{u,i}$, where i is the one color not yet unassigned.

What is the reason clause in case of a conflict?

$$\bigvee_{1 \leq \ell \leq k} c_{u,\ell}$$

Propagations (2)

Whenever we are notified of positive assignment to $c_{u,\ell}$ we propagate:

- $\neg c_{v,\ell}$ for all neighbors v of u.
- The reason is $\neg c_{u,\ell} \vee \neg c_{v,\ell}$

- $\neg c_{u,i}$ for all $1 \leq i \leq k, i \neq \ell$.
- The reason is $\neg c_{u,\ell} \vee \neg c_{u,i}$.

Notes

We can decide what to propagate on positive assignments. This also means that in case of a conflict, we can often pick the conflict.

Is this propagator useful?

Notes

We can decide what to propagate on positive assignments. This also means that in case of a conflict, we can often pick the conflict.

Is this propagator useful?

We do not really save propagations, as we almost emulate the SAT solver unit propagation. Counting the negative assignments might be faster, but probably not by much.

Can we make it useful?

The propagator allows for further improvements, e.g., symmetry breaking.

Conclusion

Conclusion

- Incremental SAT solving allows us to query and incrementally extend our formula.
- User Propagators allow us to directly interact with the SAT solver during solving.
- IPASIR-UP gives a standardized API for user propagators.
- Downside: It does not cover all use cases (yet), e.g., CP solvers.

Exercise

Exercise

We discussed that the first encoding has $O(n^3)$ many clauses (and limiting the number of colors adds even more clauses). For large graphs this means we might not even be able to generate all clauses and unnecessary clauses generally slow down solving.
We address this issue in this exercise, by using a user propagator to manually propagate variable values and only add those clauses required.
All required changes are in *encoding_propagator.h*. You only need to edit the marked part in *cb_add_reason_clause_lit()* and set the variable *_reason_clause* correctly.
However, understanding the propagation itself might help with this task. Hence, we added TODOs at the interesting spots.

- build/coloring -e 1 graphs/myciel5.col.bz2 5
- build/coloring -e 3 graphs/myciel5.col.bz2 5
- build/coloring -e 1 graphs/latin_square_10.col.bz2 9
- build/coloring -e 3 graphs/latin_square_10.col.bz2 9
- build/coloring -e 1 graphs/queen8_8.col.bz2 8
- build/coloring -e 3 graphs/queen8_8.col.bz2 8

Part V

Overall Conclusion

Conclusion

Overall, we have talked about SAT solving, both how it works internally and how to control it

We (SAT solver developer) try hard to make the solvers fast, but we have choices to make on the amount of time we target. SAT Comp: 5000s, 1000s similar, but 10s different

User propagators are more work but stronger CaDiCaL and a minisat variant by a student .. also exist for z3

The Next 10 Years?

The SAT community is small and getting older

1 year sufficient for new solver though

AI did not hit SAT solvers very strongly now

only worked within families

The Next 10 Years?

The SAT community is small and getting older — 1 year sufficient for new solver though

AI did not hit SAT solvers very strongly now — only worked within families

LLMs is hitting SAT solvers very strongly now — partially due to testing

But (as of July), no ground-breaking new idea — NVidia: no source code

The Next 10 Years?

Making solvers more useful for our users

multiple conflicts?

Most progress in the last 10 years, did not translate to improvement for users

Future: more control? or more automatic configuration?

References

- Biere, Armin et al. (2024a). “Clausal Congruence Closure”. In: *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, Pune, India, August 21-24, 2024*. Ed. by Supratik Chakraborty and Jie-Hong Roland Jiang. Vol. 305. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:25. DOI: 10.4230/LIPICS.SAT.2024.6. URL: https://doi.org/10.4230/LIPIcs.SAT.2024.6.
- — (2024b). “Clausal Equivalence Sweeping”. In: *Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024*. Ed. by Nina Narodytska and Philipp Rümmer. IEEE, pp. 1–6. DOI: 10.34727/2024/ISBN.978-3-85448-065-5_29. URL: https://doi.org/10.34727/2024/isbn.978-3-85448-065-5%5C_29.


Chowdhury, Md. Solimul, Cayden R. Codel, and Marijn J. H. Heule (2023). "A Linear Weight Transfer Rule for Local Search". In: *NASA Formal Methods - 15th International Symposium, NFM 2023, Houston, TX, USA, May 16-18, 2023, Proceedings*. Ed. by Kristin Yvonne Rozier and Swarat Chaudhuri. Vol. 13903. Lecture Notes in Computer Science. Springer, pp. 447–463. DOI: 10.1007/978-3-031-33170-1_27. URL: https://doi.org/10.1007/978-3-031-33170-1%5C_27.


— (2024). "TaSSAT: Transfer and Share SAT". In: *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*. Ed. by Bernd Finkbeiner and Laura Kovács. Vol. 14570. Lecture Notes in Computer Science. Springer, pp. 34–42. DOI: 10.1007/978-3-031-57246-3_3. URL: https://doi.org/10.1007/978-3-031-57246-3%5C_3.

- Cruz-Filipe, Luís et al. (2017). "Efficient Certified RAT Verification". In: *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*. Ed. by Leonardo de Moura. Vol. 10395. Lecture Notes in Computer Science. Springer, pp. 220–236. DOI: 10.1007/978-3-319-63046-5_14. URL: https://doi.org/10.1007/978-3-319-63046-5%5C_14.
- Eén, Niklas and Armin Biere (2005). "Effective Preprocessing in SAT Through Variable and Clause Elimination". In: *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*. Ed. by Fahiem Bacchus and Toby Walsh. Vol. 3569. Lecture Notes in Computer Science. Springer, pp. 61–75. DOI: 10.1007/11499107_5. URL: https://doi.org/10.1007/11499107%5C_5.

- Fazekas, Katalin, Armin Biere, and Christoph Scholl (2019). "Incremental Inprocessing in SAT Solving". In: *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*. Ed. by Mikolás Janota and Inês Lynce. Vol. 11628. Lecture Notes in Computer Science. Springer, pp. 136–154. DOI: 10.1007/978-3-030-24258-9_9. URL: https://doi.org/10.1007/978-3-030-24258-9%5C_9.
- Fazekas, Katalin, Aina Niemetz, et al. (2024). "Satisfiability Modulo User Propagators". In: *J. Artif. Intell. Res.* 81, pp. 989–1017. DOI: 10.1613/JAIR.1.16163. URL: https://doi.org/10.1613/jair.1.16163.
- Fazekas, Katalin, Florian Pollitt, et al. (2024). "Certifying Incremental SAT Solving". In: *LPAR 2024: Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning, Port Louis, Mauritius, May 26-31, 2024*. Ed. by Nikolaj S. Bjørner, Marijn Heule, and Andrei Voronkov. Vol. 100. EPiC Series in Computing. EasyChair, pp. 321–340. DOI: 10.29007/PDCC. URL: https://doi.org/10.29007/pdcc.

- Fazekas, Katalin, Florian Pollitt, et al. (2025). “Incremental Inprocessing Rules beyond Resolution”. In: *Joint Proceedings of the 23rd International Workshop on Satisfiability Modulo Theories and the 16th Pragmatics of SAT International Workshop co-located with the 31st International Conference on Principles and Practice of Constraint Programming, the 28th International Conference on Theory and Applications of Satisfiability Testing and the 18th International Symposium on Combinatorial Search (CP 2025 & SAT 2025 & SoCS 2025), Glasgow, UK, August 10-11, 2025*. Ed. by Jochen Hoenicke et al. Vol. 4008. CEUR Workshop Proceedings. CEUR-WS.org, pp. 190–200. URL: https://ceur-ws.org/Vol-4008/POS%5C_paper11.pdf.
- Fleury, Mathias and Armin Biere (2022). “Scalable Proof Producing Multi-Threaded SAT Solving with Gimsatul through Sharing instead of Copying Clauses”. In: *CoRR* abs/2207.13577. DOI: 10.48550/arXiv.2207.13577. arXiv: 2207.13577. URL: https://doi.org/10.48550/arXiv.2207.13577.

- Gocht, Stephan, Ciaran McCreesh, and Jakob Nordström (2020). “Subgraph Isomorphism Meets Cutting Planes: Solving With Certified Solutions”. In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*. Ed. by Christian Bessiere. ijcai.org, pp. 1134–1140. DOI: 10.24963/IJCAI.2020/158. URL: https://doi.org/10.24963/ijcai.2020/158.
- Järvisalo, Matti, Marijn Heule, and Armin Biere (2012). “Inprocessing Rules”. In: *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*. Ed. by Bernhard Gramlich, Dale Miller, and Uli Sattler. Vol. 7364. Lecture Notes in Computer Science. Springer, pp. 355–370. DOI: 10.1007/978-3-642-31365-3_28. URL: https://doi.org/10.1007/978-3-642-31365-3%5C_28.

- Nieuwenhuis, Robert, Albert Oliveras, and Cesare Tinelli (2004). "Abstract DPLL and Abstract DPLL Modulo Theories". In: *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004, Montevideo, Uruguay, March 14-18, 2005, Proceedings*. Ed. by Franz Baader and Andrei Voronkov. Vol. 3452. Lecture Notes in Computer Science. Springer, pp. 36–50. DOI: 10.1007/978-3-540-32275-7_3. URL: https://doi.org/10.1007/978-3-540-32275-7%5C_3.
- Paxian, Tobias and Armin Biere (2026). "MaxSAT Fuzzing and Delta Debugging". In: *J. Artif. Intell. Res.* 85. DOI: 10.1613/JAIR.1.19716. URL: https://doi.org/10.1613/jair.1.19716.
- Ryan, Lawrence (2004). "Efficient algorithms for clause-learning SAT solvers". MA thesis. Simon Fraser University. URL: https://www.cs.sfu.ca/~mitchell/papers/ryan-thesis.ps.

- Silva, João P. Marques and Karem A. Sakallah (1996). “GRASP - a new search algorithm for satisfiability”. In: *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996*. Ed. by Rob A. Rutenbar and Ralph H. J. M. Otten. IEEE Computer Society / ACM, pp. 220–227. DOI: 10.1109/ICCAD.1996.569607. URL: https://doi.org/10.1109/ICCAD.1996.569607.
- Stallman, Richard M. and Gerald J. Sussman (1977). “Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis”. In: *Artif. Intell.* 9.2, pp. 135–196. DOI: 10.1016/0004-3702(77)90029-7. URL: https://doi.org/10.1016/0004-3702(77)90029-7.

Weidenbach, Christoph (2015). “Automated Reasoning Building Blocks”. In: *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*. Ed. by Roland Meyer, André Platzer, and Heike Wehrheim. Vol. 9360. Lecture Notes in Computer Science. Springer, pp. 172–188. DOI: 10.1007/978-3-319-23506-6_12. URL: https://doi.org/10.1007/978-3-319-23506-6%5C_12.