CrystalBall: Gazing into the Future of SAT Solving

Mate Soos and Kuldeep S. Meel

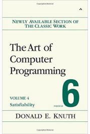
School of Computing, National University of Singapore

Collaborators: Raghav Kulkarni and Adam Chai
First Paper: In Proc. of SAT-19
Second Paper: ~~2021~~, ~~2022~~, 2023(?)

Code: https://meelgroup.github.io/crystalball/
All the code (including based on unpublished work) is available publicly.

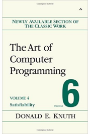
The Tale of Triumph of SAT Solvers

Modern SAT solvers are able to deal routinely with practical problems that involve many thousands of variables, although such problems were regarded as hopeless just a few years ago. (Donald Knuth, 2016)



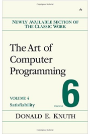
The Tale of Triumph of SAT Solvers

Modern SAT solvers are able to deal routinely with practical problems that involve many thousands of variables, although such problems were regarded as hopeless just a few years ago. (Donald Knuth, 2016)



Industrial usage of SAT Solvers: hardware verification, planning, Genome Rearrangement, Telecom Feature Subscription, Resource Constrained Scheduling, Noise Analysis, Games, $\cdots$

Modern SAT solvers are able to deal routinely with practical problems that involve many thousands of variables, although such problems were regarded as hopeless just a few years ago. (Donald Knuth, 2016)



Industrial usage of SAT Solvers: hardware verification, planning, Genome Rearrangement, Telecom Feature Subscription, Resource Constrained Scheduling, Noise Analysis, Games, $\cdots$

The story of CDCL Solvers!

$(\bar{a} \vee \bar{b}) \wedge (\bar{z} \vee b) \wedge (\bar{x} \vee \bar{z} \vee a) \wedge (y \vee b)$

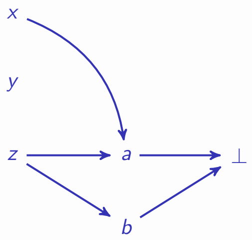
Level Dec. Unit Prop.

0 $\emptyset$

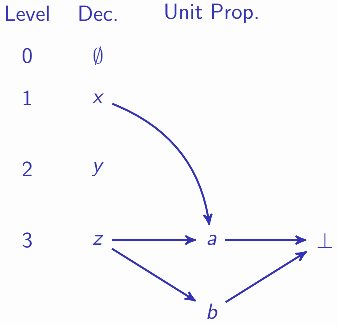
1

2

3



$(\bar{a} \vee \bar{b}) \wedge (\bar{z} \vee b) \wedge (\bar{x} \vee \bar{z} \vee a) \wedge (y \vee b)$

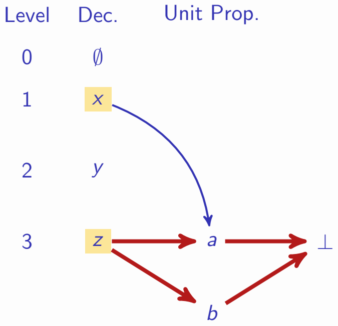


- Analyze conflict

[MSS96a,MSS96b,MSS96c,MSS96d,MSS99]

Clause learning

$(\bar{a} \vee \bar{b}) \wedge (\bar{z} \vee b) \wedge (\bar{x} \vee \bar{z} \vee a) \wedge (y \vee b)$

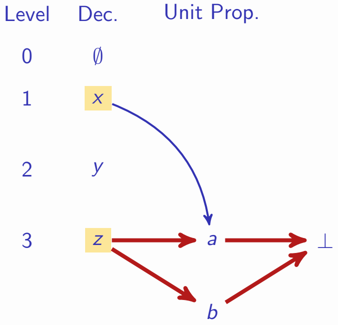


- Analyze conflict
  - Reasons: x and z

[MSS96a,MSS96b,MSS96c,MSS96d,MSS99]

Clause learning

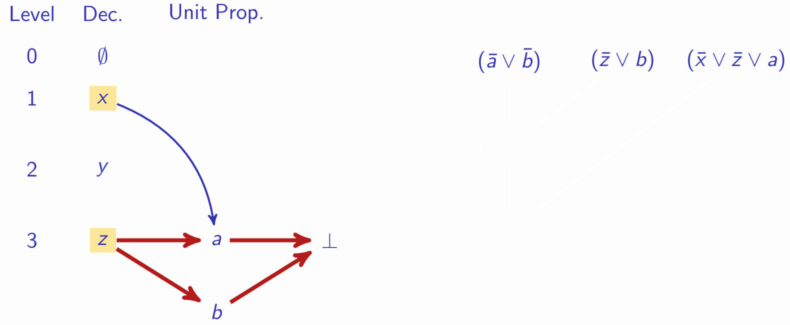
$(\bar{a} \vee \bar{b}) \wedge (\bar{z} \vee b) \wedge (\bar{x} \vee \bar{z} \vee a) \wedge (y \vee b)$



- Analyze conflict
  - Reasons: x and z
  - Create **new** clause: $(\bar{x} \vee \bar{z})$

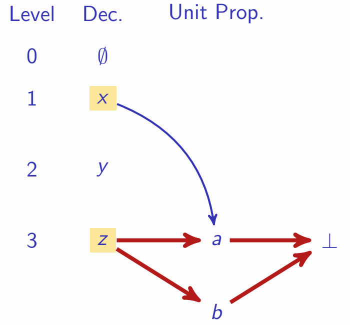
[MSS96a,MSS96b,MSS96c,MSS96d,MSS99]

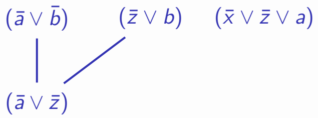
$(\bar{a} \vee \bar{b}) \wedge (\bar{z} \vee b) \wedge (\bar{x} \vee \bar{z} \vee a) \wedge (y \vee b)$



- Analyze conflict [MSS96a,MSS96b,MSS96c,MSS96d,MSS99]
  - Reasons: x and z
  - Create **new** clause: $(\bar{x} \vee \bar{z})$
- Can relate clause learning with resolution

$(\bar{a} \vee \bar{b}) \wedge (\bar{z} \vee b) \wedge (\bar{x} \vee \bar{z} \vee a) \wedge (y \vee b)$





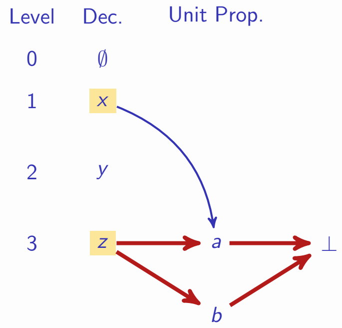
- Analyze conflict

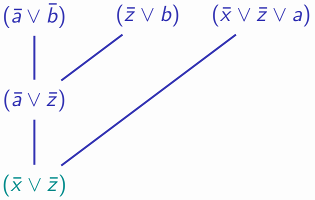
  [MSS96a,MSS96b,MSS96c,MSS96d,MSS99]

  - Reasons: x and z
  - Create **new** clause: $(\bar{x} \vee \bar{z})$
- Can relate clause learning with resolution

Clause learning

$(\bar{a} \vee \bar{b}) \wedge (\bar{z} \vee b) \wedge (\bar{x} \vee \bar{z} \vee a) \wedge (y \vee b)$





- Analyze conflict [MSS96a,MSS96b,MSS96c,MSS96d,MSS99]
  - Reasons: x and z
  - Create **new** clause: $(\bar{x} \vee \bar{z})$
- Can relate clause learning with resolution

The World of SAT Solving

- The Paradox: SAT is NP-complete yet solvers can solve problems involving millions of variables
- We understand very little why SAT solvers work!
  - Designing solvers is very hard
  - And it demands hundreds of hours (per expert deliver) every year
  - Analyze problems, find patterns, and improve heuristics

The World of SAT Solving

- The Paradox: SAT is NP-complete yet solvers can solve problems involving millions of variables
- We understand very little why SAT solvers work!
  - Designing solvers is very hard
  - And it demands hundreds of hours (per expert deliver) every year
  - Analyze problems, find patterns, and improve heuristics
- *A framework to aid the developer to understand and synthesize algorithmic heuristics for modern* SAT *solvers?*

The World of SAT Solving

- The Paradox: SAT is NP-complete yet solvers can solve problems involving millions of variables
- We understand very little why SAT solvers work!
  - Designing solvers is very hard
  - And it demands hundreds of hours (per expert deliver) every year
  - Analyze problems, find patterns, and improve heuristics
- *A framework to aid the developer to understand and synthesize algorithmic heuristics for modern* SAT *solvers?*
- CrystalBall
  - Do not intend to replace experts
  - We envision a expert in loop framework

A project born in 2018 with a 10 year horizon

Funding acknowledgment: Defense Service Organization

Data-Driven Approach to SAT Solver Design

- View SAT solvers as composition of prediction engines
  - Branching
  - Clause learning
  - Memory management
  - Restarts
- Use ML to learn and optimize behavior of prediction engines

Data-Driven Approach to SAT Solver Design

- View SAT solvers as composition of prediction engines
  - Branching
  - Clause learning
  - Memory management
  - Restarts
- Use ML to learn and optimize behavior of prediction engines
- The first step: memory management aka learnt clause deletion

The curse of learnt clauses

- Learnt clauses are very useful
- But consume memory and can slowdown other components

The curse of learnt clauses

- Learnt clauses are very useful
- But consume memory and can slowdown other components
- Delete larger clauses [E.g. MSS96a,MSS99]
- Delete less used clauses [E.g. GN02,ES03]
- Delete clauses based on Literal block distance [AS09]

The curse of learnt clauses

- Learnt clauses are very useful
- But consume memory and can slowdown other components
- Delete larger clauses [E.g. MSS96a,MSS99]
- Delete less used clauses [E.g. GN02,ES03]
- Delete clauses based on Literal block distance [AS09]

Three tiered model

- Tier 0
  - Stores learnt clauses with LBD ≤ 4
  - No cleaning is performed
- Tier 1
  - A new clause is put in Tier 1
  - if a clause C has not been used in the past 30K conflicts then the clause is moved to *Tier 2*
- Tier 2
  - Every 10K conflict, half of the clauses are cleaned.

CrystalBall Architecture

Architecture

- For inference, we want to do supervised learning
- For every clause, we need values of different features and a label
- The inference engine should learn the model to predict the label

Architecture

- For inference, we want to do supervised learning
- For every clause, we need values of different features and a label
- The inference engine should learn the model to predict the label

Components of CrystalBall

1. Feature Engineering

Architecture

- For inference, we want to do supervised learning
- For every clause, we need values of different features and a label
- The inference engine should learn the model to predict the label

Components of CrystalBall

1. Feature Engineering
2. Labeling

Architecture

- For inference, we want to do supervised learning
- For every clause, we need values of different features and a label
- The inference engine should learn the model to predict the label

Components of CrystalBall

1. Feature Engineering
2. Labeling
3. Data collection

Architecture

- For inference, we want to do supervised learning
- For every clause, we need values of different features and a label
- The inference engine should learn the model to predict the label

Components of CrystalBall

1. Feature Engineering
2. Labeling
3. Data collection
4. Inference Engine

- Global features: property of the CNF formula at the time of genesis
- Contextual features: computed at the time of generation of the clause and relate to the generated clause, e.g. LBD score
- Restart features: correspond to statistics (average and variance) on the size and LBD of clauses, branch depth, trail depth during the current and previous restart.
- Performance features: performance parameters of the learnt clause such as the number of times the solver played part of a 1stUIP conflict clause generation

Total # of features: 127

- Attempt #1: For a learnt clause C in memory, can we predict every 10K conflicts if C will be used in future?
  - But not every learnt clause is useful eventually!

Part2: Labeling

- Attempt #1: For a learnt clause C in memory, can we predict every 10K conflicts if C will be used in future?
  - But not every learnt clause is useful eventually!
  - What if C is used in future to derive clause D, which is never used in future.
- Attempt #2: For a learnt clause C in memory, can we predict every 10K conflicts if C will be used in future for derivation of a *useful* clause?
  - How do we define a useful clause?

- We focus on UNSAT formulas
  - SAT solver can be viewed as trying to find the proof of unsatisfiability. When the formula is satisfiable, it *discovers* satisfiable assignments.

- We focus on UNSAT formulas
  - SAT solver can be viewed as trying to find the proof of unsatisfiability. When the formula is satisfiable, it *discovers* satisfiable assignments.
- A clause is useful if it is involved in the final UNSAT proof.

- We focus on UNSAT formulas
  - SAT solver can be viewed as trying to find the proof of unsatisfiability. When the formula is satisfiable, it *discovers* satisfiable assignments.
- A clause is useful if it is involved in the final UNSAT proof.
- For some cases, more than $> 50\%$ clauses are useful

- We focus on UNSAT formulas
  - SAT solver can be viewed as trying to find the proof of unsatisfiability. When the formula is satisfiable, it *discovers* satisfiable assignments.
- A clause is useful if it is involved in the final UNSAT proof.
- For some cases, more than $> 50\%$ clauses are useful
- But we can only keep less than 5% of clauses in memory

- We focus on UNSAT formulas
  - SAT solver can be viewed as trying to find the proof of unsatisfiability. When the formula is satisfiable, it *discovers* satisfiable assignments.
- A clause is useful if it is involved in the final UNSAT proof.
- For some cases, more than $> 50\%$ clauses are useful
- But we can only keep less than 5% of clauses in memory
  Need to consider temporal aspect of usefulness
- We associate a counter with execution of SAT solver: incremented with every conflict
- expiry (C): The value of counter when C was last used in the UNSAT proof

- We focus on UNSAT formulas
  - SAT solver can be viewed as trying to find the proof of unsatisfiability. When the formula is satisfiable, it *discovers* satisfiable assignments.
- A clause is useful if it is involved in the final UNSAT proof.
- For some cases, more than $> 50\%$ clauses are useful
- But we can only keep less than 5% of clauses in memory
  Need to consider temporal aspect of usefulness
- We associate a counter with execution of SAT solver: incremented with every conflict
- expiry (C): The value of counter when C was last used in the UNSAT proof
- A clause is useful in future at t if expiry(C) $> t$.

Part 3: Data Collection

- Just record the trace of the solver

Part 3: Data Collection

- Just record the trace of the solver
- Works well for toy benchmarks.

- Just record the trace of the solver
- Works well for toy benchmarks.
- We are interested in understanding performance for competition benchmarks – large benchmarks

Part 3: Data Collection

- Just record the trace of the solver
- Works well for toy benchmarks.
- We are interested in understanding performance for competition benchmarks – large benchmarks
- Need to reconstruct *approximate/inexact* trace

Part 3: Data Collection

- Just record the trace of the solver
- Works well for toy benchmarks.
- We are interested in understanding performance for competition benchmarks – large benchmarks
- Need to reconstruct *approximate/inexact* trace drat-trim.

- Forward pass
  - The solver keeps track of features of each clause and dumps all the learnt clauses after we reach UNSAT.
  - genesis(C): The value of counter when C was learnt
  - expiry (C): The value of counter when C was last used in the UNSAT proof

- Forward pass
  - The solver keeps track of features of each clause and dumps all the learnt clauses after we reach UNSAT.
  - genesis(C): The value of counter when C was learnt
  - expiry (C): The value of counter when C was last used in the UNSAT proof
- Backward pass
  - DRAT-trim is used to reconstruct the proof while satisfying the constraint while satisfying the constraint expiry(C) $>$ genesis(C).

- Consider an UNSAT formula φ defined as:

$$\varphi := (\neg d \vee \neg g \vee f) \wedge (\neg d \vee \neg g \vee \neg f) \wedge (\neg d \vee g) \wedge (a \vee \neg c \vee d) \\ \wedge (\neg a \vee \neg c \vee d) \wedge (g) \wedge (c \vee d \vee \neg g)$$

- One possible execution of the solver can produce the following learnt clauses $\{(\neg d \vee \neg g), (c \vee \neg g), (c), (\neg d), (a \vee \neg c), (\neg c \vee d), (\neg c \vee \neg g), (\neg c)\}$.

DRAT-based Labeling

The clause of φ as "red".

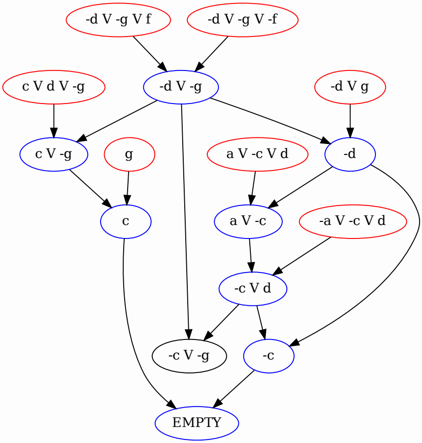


Figure: Proof Generated by DRAT-Trim

- Why not keep track of the proof during forward pass?
  - We want to handle SAT competition benchmarks for a state of the art solver (CryptoMiniSAT) and keeping track of full trace is infeasible
  - There is no reason to believe that we should try to optimize clause deletion for the proof generated by solver.
  - Game-theoretic view A better clause deletion may lead to a better proof, so using an external optimized proof generator may be a better idea.

- XGBoost for final working model
- 400 unsatisfiable instances from the SAT Competitions (2014-20)
- Trained on 216 files that were solved with CryptoMiniSat
- Usage of multi-tiered structure in modern SAT solvers

Preliminary Insights

Testing on SAT instances

- 400 instances from SAT competition

	Solved Instances	PAR-2 Score	Time spent in Clause cleaning
cms-default	255	4502	0.3%
cms-crystalball	256	4512	7.5%

- cms-crystalball uses 34% less clauses in-memory on average

Benchmark Generation (Grain Cipher)

- randomly generated key, plaintext, and correct ciphertext
- CNF formula over ciphertext and the plaintext so that satisfying assignment is key
- Set $N \in [94, 99]$ bits randomly, therefore, unsatisfiable with high probability

Runtime Performance: Grain

Solver	Solved	PAR-2 score	Clause deletion time
cms-default	25	5226.6	0.4%
cms-crystalball	66	4920.4	10.4%

Table: The default and the crystalball-based CryptoMiniSat solving 120 randomly generated Grain cipher benchmarks

The power of interpretable classifiers: Feature Ranking

The power of interpretable classifiers: Feature Ranking

1. Used during UIP1 generation per round (i.e. per 10k/15k/25k), and total/time-in-solver
2. Used for propagating per round (i.e. per 10k/15k/25k), and total/time-in-solver
3. LBD
4. Relative decile of clause since last restart with respect to propagation usage
5. Relative decile clause this round with respect to 1-UIP

Summary

- Data-driven insights for SAT solving
- Allows us to handle competition benchmarks
- Preliminary results demonstrate the power of data-driven approach

More Open Questions than Answers

- Democratize the design of solvers; allows people without expertise in SAT solving to test out their ideas
  - Working on setting up a NeurIPS challenge
  - Python module release
- Interface for other solvers
- Extend CrystalBall for branching, clause learning, and restarts

Join us: `https://meelgroup.github.io/crystalball/`
All the code (including based on unpublished work) is available publicly.