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All the code (including based on unpublished work) is available publicly.
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, ···
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, ···

The story of CDCL Solvers!
Clause learning

$$(\overline{a} \lor b) \land (\overline{z} \lor b) \land (\overline{x} \lor \overline{z} \lor a) \land (y \lor b)$$

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$x$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$y$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$z$</td>
<td>$a$</td>
</tr>
</tbody>
</table>

- Analyze conflict
  - Reasons: $x$ and $z$
  - Create new clause: $(\overline{x} \lor \overline{z})$

- Can relate clause learning with resolution
Clause learning

\[(\overline{a} \lor \overline{b}) \land (\overline{z} \lor b) \land (\overline{x} \lor \overline{z} \lor a) \land (y \lor b)\]

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\emptyset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>a \rightarrow \bot</td>
</tr>
</tbody>
</table>

- Analyze conflict

[Ref: MSS96a, MSS96b, MSS96c, MSS96d, MSS99]
Clause learning

\[(\overline{a} \lor \overline{b}) \land (\overline{z} \lor b) \land (\overline{x} \lor \overline{z} \lor a) \land (y \lor b)\]

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$x$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$y$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$z \rightarrow a \rightarrow \bot \rightarrow b$</td>
<td></td>
</tr>
</tbody>
</table>

- Analyze conflict
  - Reasons: $x$ and $z$

[MSS96a, MSS96b, MSS96c, MSS96d, MSS99]
Clause learning

\[(\overline{a} \lor \overline{b}) \land (\overline{z} \lor b) \land (\overline{x} \lor \overline{z} \lor a) \land (y \lor b)\]

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$x$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$y$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$z$</td>
<td>$a$</td>
</tr>
</tbody>
</table>

• Analyze conflict
  - Reasons: $x$ and $z$
  - Create **new** clause: $(\overline{x} \lor \overline{z})$

[MS96a,MS96b,MS96c,MS96d,MS99]
Clause learning

\[(\bar{a} \lor \bar{b}) \land (\bar{z} \lor b) \land (\bar{x} \lor \bar{z} \lor a) \land (y \lor b)\]

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\emptyset)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(x)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(y)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(z) → (a) → (\bot)</td>
<td>((\bar{a} \lor \bar{b})) ((\bar{z} \lor b)) ((\bar{x} \lor \bar{z} \lor a))</td>
</tr>
</tbody>
</table>

- Analyze conflict
  - Reasons: \(x\) and \(z\)
  - Create **new** clause: \((\bar{x} \lor \bar{z})\)
- Can relate **clause learning** with resolution
Clause learning

\[(\bar{a} \lor \bar{b}) \land (\bar{z} \lor b) \land (\bar{x} \lor \bar{z} \lor a) \land (y \lor b)\]

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(x)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(y)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(z)</td>
<td>(a) → (\perp)</td>
</tr>
</tbody>
</table>

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

• Can relate clause learning with resolution

[\text{MSS96a, MSS96b, MSS96c, MSS96d, MSS99}]
## Clause learning

\[(\bar{a} \lor \bar{b}) \land (\bar{z} \lor b) \land (\bar{x} \lor \bar{z} \lor a) \land (y \lor b)\]

<table>
<thead>
<tr>
<th>Level</th>
<th>Dec.</th>
<th>Unit Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(x)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(y)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(z)</td>
<td>(\perp)</td>
</tr>
</tbody>
</table>

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

- Can relate clause learning with resolution

Reference: [MSS96a,MSS96b,MSS96c,MSS96d,MSS99]
• **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 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 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
• 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
- Delete less used clauses
- Delete clauses based on Literal block distance

[E.g. MSS96a,MSS99]
[E.g. GN02,ES03]
[AS09]
The curse of learnt clauses

- Learnt clauses are very useful
- But consume memory and can slowdown other components
- Delete larger clauses
- Delete less used clauses
- Delete clauses based on Literal block distance

Three tiered model

- Tier 0
  - Stores learnt clauses with \( LBD \leq 4 \)
  - 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
• 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
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
• 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
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
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
Part 1: Feature Engineering

- 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!
• **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?
Part2: Labeling
Useful Clauses

- 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
Part2: Labeling

Useful Clauses

- 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

Works well for toy benchmarks. We are interested in understanding performance for competition benchmarks – large benchmarks. Need to reconstruct approximate/inexact drat-trim.
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 \textit{approximate/inexact} trace \texttt{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
Part 3: Data Collection

• 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 $\varphi$ defined as:

$$
\varphi := (\neg d \lor \neg g \lor f) \land (\neg d \lor \neg g \lor \neg f) \land (\neg d \lor g) \land (a \lor \neg c \lor d) \\
\land (\neg a \lor \neg c \lor d) \land (g) \land (c \lor d \lor \neg g)
$$

• One possible execution of the solver can produce the following learnt clauses

$$
\{(\neg d \lor \neg g), (c \lor \neg g), (c), (\neg d), (a \lor \neg c), (\neg c \lor d), (\neg c \lor \neg g), (\neg c)\}.
$$
DRAT-based Labeling

The clause of $\varphi$ 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.
Part 4: Training and Testing
How to use predictions

- 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

<table>
<thead>
<tr>
<th></th>
<th>Solved Instances</th>
<th>PAR-2 Score</th>
<th>Time spent in Clause cleaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>cms-default</td>
<td>255</td>
<td>4502</td>
<td>0.3%</td>
</tr>
<tr>
<td>cms-crystalball</td>
<td>256</td>
<td>4512</td>
<td>7.5%</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>Solver</th>
<th>Solved</th>
<th>PAR-2 score</th>
<th>Clause deletion time</th>
</tr>
</thead>
<tbody>
<tr>
<td>cms-default</td>
<td>25</td>
<td>5226.6</td>
<td>0.4%</td>
</tr>
<tr>
<td>cms-crystalball</td>
<td>66</td>
<td>4920.4</td>
<td>10.4%</td>
</tr>
</tbody>
</table>

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