CrystalBall: Gazing in the Black Box of SAT Solving

Mate Soos[1], Kuldeep S. Meel[1], and **Raghav Kulkarni**[2]

[1]School of Computing, National University of Singapore

[2]Chennai Mathematical Institute

Several open positions for post-docs and PhD students in the world's best city for expats to live (Singapore): Amazing food, sun all year around, and low taxes

The Price of Success

- SAT is still NP-complete yet solvers tend to solve problems involving millions of variables
- The solvers of today are very complex
- We understand very little why SAT solvers work!

The Price of Success

- SAT is still NP-complete yet solvers tend to solve problems involving millions of variables
- The solvers of today are very complex
- We understand very little why SAT solvers work!
- 50,000 hours of CPU time plus tens of human hours tuning parameters in CryptoMiniSAT for 2018 competition (won third place in SAT 2018 competition)

Data-Driven Design of SAT solver

- View SAT solvers as composition of prediction engines
  - Branching
  - Clause learning
  - Memory management
  - Restarts

Data-Driven Design of SAT solver

- View SAT solvers as composition of prediction engines
  - Branching
  - Clause learning
  - Memory management
  - Restarts
- Prior Work
  - Machine learning to optimize behavior of prediction engines
  - Focused on using runtime or proxy for runtime

Data-Driven Design of SAT solver

- View SAT solvers as composition of prediction engines
  - Branching
  - Clause learning
  - Memory management
  - Restarts
- Prior Work
  - Machine learning to optimize behavior of prediction engines
  - Focused on using runtime or proxy for runtime

Whether it is possible to develop a framework to provide white-box access to execution of SAT *solver, which can aid the developer to understand and synthesize algorithmic heuristics for modern* SAT *solvers?*

Data-Driven Design of SAT solver

- View SAT solvers as composition of prediction engines
  - Branching
  - Clause learning
  - Memory management
  - Restarts
- Prior Work
  - Machine learning to optimize behavior of prediction engines
  - Focused on using runtime or proxy for runtime
- CrystalBall *Whether it is possible to develop a framework to provide white-box access to execution of* SAT *solver, which can aid the developer to understand and synthesize algorithmic heuristics for modern* SAT *solvers?*

Data-Driven Design of SAT solver

- View SAT solvers as composition of prediction engines
  - Branching
  - Clause learning
  - Memory management
  - Restarts
- Prior Work
  - Machine learning to optimize behavior of prediction engines
  - Focused on using runtime or proxy for runtime
- CrystalBall *Whether it is possible to develop a framework to provide white-box access to execution of* SAT *solver, which can aid the developer to understand and synthesize algorithmic heuristics for modern* SAT *solvers?*
- What CrystalBall is not about?
  - Replacing experts

Data-Driven Design of SAT solver

- View SAT solvers as composition of prediction engines
  - Branching
  - Clause learning
  - Memory management
  - Restarts
- Prior Work
  - Machine learning to optimize behavior of prediction engines
  - Focused on using runtime or proxy for runtime
- CrystalBall *Whether it is possible to develop a framework to provide white-box access to execution of* SAT *solver, which can aid the developer to understand and synthesize algorithmic heuristics for modern* SAT *solvers?*
- What CrystalBall is not about?
  - Replacing experts
- We envision a expert in loop framework

Data-Driven Design of SAT solver

- View SAT solvers as composition of prediction engines
  - Branching
  - Clause learning
  - Memory management
  - Restarts
- Prior Work
  - Machine learning to optimize behavior of prediction engines
  - Focused on using runtime or proxy for runtime
- CrystalBall *Whether it is possible to develop a framework to provide white-box access to execution of* SAT *solver, which can aid the developer to understand and synthesize algorithmic heuristics for modern* SAT *solvers?*
- What CrystalBall is not about?
  - Replacing experts
- We envision a expert in loop framework
- As a first step, we have focused on memory management: learnt clause deletion. All models are wrong. Some are useful.

The curse of learnt clauses

- Learnt clauses are very useful
- But they consume memory and can slowdown other components of SAT solving
- Not practical to keep all the learnt clauses
- Delete larger clauses [E.g. MSS96a,MSS99]
- Delete less used clauses [E.g. GN02,ES03]
- Delete clauses based on Literal block distance [AS09]

Clause Deletion

Three tiered model

- Tier 0
  - Stores learnt clauses with LBD ≤ 4
  - LBD of a clause is the number of different decision levels corresponding to the literals in the learnt clause
  - 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

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

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.

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: 212

- Ideal: the scale of features is independent of the problem
- Relativize the feature values by taking average feature values in the history as a guideline and measuring the ratio of the actual feature value and this average instead.

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!

- 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
- Useful A clause is useful in future at t if expiry(C) $> t$.

- 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
- Useful A clause is useful in future at t if expiry(C) $> t$.
- Can we predict every 10K conflicts for a clause C if C will be useful in future?

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.

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

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

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).
  - Key modifications
    - For every clause we attach a unique ID to every clause as the same clause can be learned twice, so it is important to track each clause
    - We supply genesis of a clause so that a clause is not used in the proof before its genesis

- 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.

- We employ standard VSIDS heuristic augmented with polarity caching

- We employ standard VSIDS heuristic augmented with polarity caching

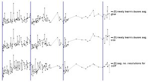
- We employ standard VSIDS heuristic augmented with polarity caching
- We disable the adaptive restart strategy. we do not want our inference to be based on the data that is potentially polluted due to the adaptive restart strategy.

- We employ standard VSIDS heuristic augmented with polarity caching
- We disable the adaptive restart strategy. we do not want our inference to be based on the data that is potentially polluted due to the adaptive restart strategy.
- We disable in-processing and perform the pre-processing. The in-processing transforms the clauses and thereby can affect the inference process.
- We keep all the learnt clauses in memory

Looking back over the years

Visualizing SAT solving

June 16, 2012 · Uncategorized · SAT, visualisation



Visualizing the solving of mizh-md5-47-3.cnf

Visualizing what happens during SAT solving has been a long-term goal of mine, and finally, I have managed to pull together something that I feel confident about. The system is fully explained in the liked image on the right, including how to read the graphs and why I made them. Here, I would like to talk about the challenges I had to overcome to create the system.

Gathering information

Gathering information during solving is challenging for two reasons. First, it's hard to know *what* to gather. Second, gathering the information should not affect overall speed of the solver (or only minimally), so the code to gather the information has to be well-written. To top it all, if much information is gathered, these have to be structured in a sane way, so it's easy to access later.

It took me about 1-1.5 months to write the code to gather all information I wanted. It took a lot of time to correctly structure and to decide about how to store/summarize the information gathered. There is much more gathered than shown on the webpage, but more about that below.

Selecting what to display, and how

This may sound trivial. Some would simply say: just display all information! But what we really want is not just plain information: what good is it to print 100'000 numbers on a screen? The data has to be displayed in a meaningful and visually understandable way.

Getting to the current layout took a lot of time and many-many discussions with all all my friends and colleagues. I am eternally grateful for their input — it's hard to know how good a layout is until someone sees it for the first time, and completely misunderstands it. Then you know you have to change it: until then, it was trivial to you what the graph meant, after all, you made it!

What to display is a bit more complex. There is a lot of data gathered, but what is interesting? Naturally, I couldn't display everything, so I had to select. But selection may become a form of misrepresentation: if some important data isn't displayed, the system is effectively lying. So, I tried to add as much as possible that still made sense. This lead to a very large table of graphs, but I think it's still under-

Machine Learning and SAT

August 9, 2015 · Development, Research, SAT · glues, lingeling, machine learning

I have lately been digging myself into a deep hole with machine learning. While doing that it occurred to me that the SAT community has essentially been trying to imitate some of ML in a somewhat poor way. Let me explain.

CryptoMiniSat and clause cleaning strategy selection

When CryptoMiniSat won the SAT Race of 2010, it was in large part because I realized that glucose at the time was essentially unable to solve cryptographic problems. I devised a system where I could detect which problems were cryptographic. It checked the activity stability of variables and if they were more stable than a threshold, it was decided that the problem was cryptographic. Cryptographic problems were then solved using a geometric restart strategy with clause activities for learnt database cleaning. Without this hack, it would have been impossible to win the competition.

It is clear that there could have been a number of ways to detect that a problem is cryptographic without using such an elaborate scheme. However, that would have demanded a mixture of more features to decide. The scheme only used the average and the standard deviation.

Lingeling and clause cleaning strategy selection

The decision made by lingeling about whether to use glues or activities to clean learnt clauses is somewhat similar to my approach above. It calculates the average and the standard deviation of the learnt clauses' glues and then makes a decision. Looking at the code, the option actavgmax/stdmin/stdmax gives the cutoffs and the function lglneedacts calculates the values and decides. This has been in lingeling since 2011 (lingeling-587f).

Probably a much better decision could be made if more data was taken into account (e.g. activities) but as a human, it's simply hard to make a decision based on more than 2-3 pieces of data.

Enter machine learning

It is clear that the above schemes were basically trying to extract some *feature* from the SAT solver and then decide what *features* (glues/activities) to use to clear the learnt clause database. It is also clear that both have been extremely effective, it's by no luck that they have been inside successful SAT solvers.

The question is, can we do better? I think yes. First of all, we don't need to cut the problem into two steps. Instead, we can integrate the features extracted from the solver (variable activities, clause glue distribution, etc) and the features from the clause (glue, activities, etc.) and make a decision whether to keep the clause or not. This means we would make keep/throwaway decisions on individual claus-

- Usage of multi-tiered structure in modern SAT solvers

- Usage of multi-tiered structure in modern SAT solvers
- *keep-short*: Mark clause for not deletion for another 10K conflicts
- *keep-long*: Mark clause for not deletion for another 100K conflicts

- Usage of multi-tiered structure in modern SAT solvers
- *keep-short*: Mark clause for not deletion for another 10K conflicts
- *keep-long*: Mark clause for not deletion for another 100K conflicts
- Since we need to make decisions every 10K/100K conflicts, suffices to predict the binary decision if expiry(C) $>$ current conflict
- Classification instead of regression!

- Two constraints
  - Our 212 features are mixed or heterogeneous.
  - No straightforward manner to normalize all of our features.
- The SVM and other linear models require carefully normalized homogeneous features.
- We chose the random forest as the classifier for our inference engine

Preliminary Insights

Experimental Setup

- All the UNSAT instances from SAT 2014-17.
- Each instance was ran with timeout of 20,000 seconds and CrystalBall finished execution for 260 instances
- The number of learnt clauses for different problems varied from few hundreds to millions
- We sampled 2000 data points from each benchmarks to ensure fair representation for each benchmark.
- We discarded 50 benchmarks that had less than 2000 data points.
- In total, we had 422K data points.
- Standard split into 70% training and 30% training.

Accuracy of Engine: *keep-short*

		Prediction	
		Throw	Keep
Ground	Throw	0.64	0.36
truth	Keep	0.11	0.89

Table: Confusion matrix for *keep-short*

Accuracy of Engine: *keep-long*

		Prediction	
		Throw	Keep
Ground truth	Throw	0.63	0.37
	Keep	0.09	0.91

Table: Confusion matrix for *keep-long*

The power of interpretable classifiers

Feature Ranking for *keep-short*

1. rdb0.used_for_uip_creation: Number of times that the conflict took part in a 1UIP conflict generation since its creation.
2. rdb0.last_touched_diff: Number of conflicts ago that the clause was used during a 1UIP conflict clause generation.
3. rdb0.activity_rel: Activity of the clause, relative to the activity of all other learned clauses at the point of time when the decision to keep or throw away the clause is made.
4. rdb0.sum_uip1_used: Number of times that the clause took part in a 1UIP conflict generation since its creation.
5. rdb1.used_for_uip_creation: Same as rdb0.used_for_uip_creation but instead of the current round, it is data from the previous round (i.e. 10k conflicts earlier)

LBD is not a top-5 feature

1. rdb0.sum_uip1_used: Number of times that the conflict took part in a 1UIP conflict generation since its creation.
2. rdb1.used_for_uip_ creation: Same as rdb0.used_for_uip_creation but instead of the current round, it is data from the previous round (i.e. 10k conflicts earlier)
3. rdb0.used_for_uip_creation: Number of times that the clause took part in a 1UIP conflict generation since its creation.
4. rdb0.act_ranking: Activity ranking of the clause (i.e. 1st, 2nd, etc.), among all learned clauses at the point of time when the decision to keep or throw away the clause is made.
5. rdb0.act_ranking_top_10: Whether the activity of the clause belongs to the top 10% among all learned clauses at the point of time when the decision to keep or throw away the clause is made.

LBD is not a top-5 feature

Beyond speedups?

AAAI-19 "Expert" Reviewer

...It is very easy to collect data, but a completely different level of performance to be able to use it to achieve a speedup. The big question after reading the paper is: so what?

An efficient Ph.D. student could have collected this data in 1-2 weeks of work.

As such, there is no contribution that is worth publishing....

[Question for Rebuttal]: Why did you submit this paper...?

Comparison with state of the art Solver

- 934 instances from SAT Competitions 2014-17 with a timeout of 5000 seconds.

Comparison with state of the art Solver

- 934 instances from SAT Competitions 2014-17 with a timeout of 5000 seconds.
- *Maple_LCM_Dist* : 591 instances (2017 winning solver)

Comparison with state of the art Solver

- 934 instances from SAT Competitions 2014-17 with a timeout of 5000 seconds.
- *Maple_LCM_Dist* : 591 instances (2017 winning solver)
- CryptoMiniSAT plus learned classifier: 612 instances
  - Solved SAT: 271
  - Solved UNSAT: 341
- The ratio of SAT to UNSAT instances is almost same to *Maple_LCM_Dist*.

Comparison with state of the art Solver

- 934 instances from SAT Competitions 2014-17 with a timeout of 5000 seconds.
- *Maple_LCM_Dist* : 591 instances (2017 winning solver)
- CryptoMiniSAT plus learned classifier: 612 instances
  - Solved SAT: 271
  - Solved UNSAT: 341
- The ratio of SAT to UNSAT instances is almost same to *Maple_LCM_Dist*.
- Training was only on UNSAT instances – shows generalizability

Conclusion

Summary

- Goal: Data-driven insights for SAT solving
- CrystalBall is scalable framework built on the state of the art solver to have whitebox access to SAT solving
- Allows us to handle competition benchmarks
- Preliminary results demonstrate the power of data-driven approach: There are several features with prediction power comparable (better?) to LBD

More Open Questions than Answers

- Democratize the design of solvers; allows researchers without deep expertise in software engineering of SAT solvers to test out their ideas
- Design new features. For derivative features, you do not even need to rerun the solver
- Learn complex models
- Extend CrystalBall for branching, clause learning, and restarts
- Interface for other solvers
- An application area for interpretable machine learning

Code: `https://meelgroup.github.io/crystalball/`