CrystalBall: Gazing in the Black Box of SAT Solving

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

[1] School of Computing, National University of Singapore
[2] Chennai Mathematical Institute, Chennai

Abstract. Boolean satisfiability is a fundamental problem in computer science with a wide range of applications including planning, configuration management, design and verification of software/hardware systems. The annual SAT competition continues to witness impressive improvements in the performance of the winning SAT solvers largely thanks to the development of new heuristics arising out of intensive collaborative research in the SAT community. Modern SAT solvers achieve scalability and robustness with sophisticated heuristics that are challenging to understand and explain. Consequently, the development of new algorithmic insights has been primarily restricted to *expert intuitions* and evaluation of the new insights have been restricted to performance measurement in terms of the runtime of solvers or a proxy for the runtime of solvers. In this context, one may ask: *whether it is possible to develop a framework to provide white-box access to the execution of SAT solver that can aid both SAT solver developers and users to synthesize algorithmic heuristics for modern SAT solvers?*
The primary focus of our project is precisely such a framework, which we call CrystalBall. More precisely, we propose to view modern *conflict-driven clause learning* (CDCL) solvers as a composition of classifiers and regressors for different tasks such as *branching, clause memory management, and restarting.* The primary objective of this paper is to introduce a framework to peek inside the SAT solvers – CrystalBall– to the AI and SAT community. The current version of CrystalBall focuses on deriving a classifier to keep or throw away a learned clause. In a departure from recent machine learning based techniques, CrystalBall employs supervised learning and uses extensive, multi-gigabyte data extracted from runs of a single SAT solver to perform predictive analytics.

1 Introduction

Boolean satisfiability is a fundamental problem in computer science with a wide range of applications including planning, configuration management, design and verification of software/hardware systems. While the mention of SAT can be traced to the early 19th century, efforts to develop practically successful SAT solvers go back to 1960s [5]. The annual SAT competition continues to witness impressive improvements in the performance of the winning SAT solvers largely thanks to the development of new heuristics arising out of intensive collaborative research in SAT community [1]. While the presence of scores of heuristics has contributed to the robustness and scalability of the SAT solvers, it has come at the cost of lack of explainability and understanding the behavior of SAT solvers. Consequently, the development of new algorithmic insights has been primarily restricted to *expert intuitions* and evaluation of the new insights has been restricted to performance measurement in terms of the runtime of solvers or a proxy for the runtime of solvers.

One of the most critical, but not well-understood heuristic in SAT solvers is learned clause database management, even though it plays a crucial role in the performance of CDCL SAT solvers. For effective database management, a SAT solver needs to decide which learned clauses to keep in the memory as this will affect both memory usage, and, more importantly, runtime - thanks to the potentially useless clause being checked for potential propagation or conflict. While there are several different heuristics used by different modern SAT solvers, the poor understanding of when learned clauses are used during the proof generation makes it hard for the SAT community to develop and improve the heuristics further.

A promising direction in better understanding heuristics in modern SAT solvers was pursued recently by Liang et al [13,16,15]. It focused on using machine learning for the development of a new variable branching heuristic. Their project, MapleSAT, achieved a significant milestone by winning two gold medals in the 2016 SAT competition and two silver medals in the 2017 SAT competition. While the success of MapleSAT shows the potential of machine learning techniques in SAT solving, the framework for designing heuristics is still primarily restricted to a black box view of SAT solving by focusing on runtime or a proxy of runtime, similarly to [12] which focuses on learning efficient heuristics for QBF formulas. In this context, one may ask: *whether it is possible to develop a framework to provide white-box access to the execution of SAT solver, which can aid the SAT solver developer to synthesize algorithmic heuristics for modern SAT solvers?*

The purpose of our project, called CrystalBall, is to answer the above question affirmatively. We view modern CDCL solvers as a composition of classifiers and regressors for different tasks such as *branching* (which variable to branch on), *clause memory management* (which learned clauses to keep in the memory and which ones to throw), *restarts* (when to terminate a branch and restart), and the like. To gain a deeper understanding of the underlying classifiers, as a first step, we have built a framework to provide white box access to SAT solvers during the solving phase. We envision that such a framework to allow the end user to gain an in-depth data-driven understanding of the performance of their heuristics and aid them in designing better heuristics. We do not aim to replace *expert intuition* but propose an *expert-in-the-loop* approach where the expert is aided with statistically sound explainable classifiers by CrystalBall.

The current version of CrystalBall focuses on inferring two classifiers to predict whether a learnt clause should be kept or thrown away. We take a supervised learning approach that required the design of a sophisticated architecture for data-collection from the execution trace of a SAT solver. We then use supervised learning to infer a set of interpretable classifiers, which are translated to C++ code[3], compiled into CryptoMiniSat and executed along with the solver. It is worth mentioning that the classifiers were inferred using only a small set of UNSAT instances. The ability of the learned classifier to handle SAT instances almost as well as UNSAT instances, along with being able to outperform the state-of-the-art solver of 2017, provides strong evidence in support for the choices of different components in our framework. CrystalBall is released as an open-source framework and we believe CrystalBall could serve as a backbone for designing algorithmic ideas for modern CDCL solvers via a data-driven understanding of their heuristics.

The rest of the paper is organized as follows. We first introduce notation and preliminaries in Section 2. We describe in detail the feature engineering, large-scale extraction of data from SAT solvers, labeling of data and classifier in Section 3 and present preliminary results in Section 4. We finally conclude in Section 5 with an outlook for future work.

2 Notations and Preliminaries

We borrow the preliminaries and terminology from [8]. Let X be the set of n Boolean variables. A literal a is either a variable x or its negation $\bar{x}$. A clause $C = a_1 \vee \ldots \vee a_k$ is a disjunction of literals. A clause of size 1 is called a unit clause. A formula F over X is in Conjunctive Normal Form (CNF) if it is a conjunction over clauses.

A resolution derivation of C from a formula F is a sequence of clauses $(C_1, C_2, \ldots C_\tau)$ such that $C_\tau = C$ and every C_i is either a clause in F (an axiom) or is derived from clauses C_j, C_k with $j, k < i$, by the propositional resolution rule, $A \vee a \diamond B \vee \bar{a} \rightarrow A \vee B$ We refer to this resolution step $\diamond$ as "$A \vee a$ and $B \vee \bar{a}$ are resolved over a". A *unit propagation* is a special propositional rule when $B = \emptyset$.

Given an input formula F, the run of a CDCL solver consists following sequence of actions:

1. Choose a variable to branch on and assign a value (0 or 1) to the chosen variable (if x_i is assigned 0 then it is equivalent to adding $\bar{x_i}$ to the set of clauses).
2. Use unit propagation rules until some clause gets falsified. This is known as *conflict*.
3. Derive a *learned clause* from the conflict, add the learned clause to the database $\mathcal{D}$ of clauses, and *backtrack*.
4. *Restart* the search after some time and start branching again from the top

When we start branching, i.e., choose a variable and assign a value to it then the literals implied by this assignment are said to belong to the first *decision*

[3] Translation of the predictor happens by recursively walking the decision tree(s) and emitting human-readable C++ code

level. Subsequently if we make another decision to branch on another variable then the new literals implied at this step are said to belong to the second decision level. A literal can belong to at most one decision level. We define *LBD* (Literal Block Distance) score of a clause to be the number of distinct decision levels to which the literals of the clause belong at the time of creation of the clause [2].

One can view modern CDCL solver as composition of classifiers and regressors to perform the following actions such as branching, learned clause cleaning, and restarting. It is worth noting that much of the prior work in the SAT community has implicitly focused on designing better classifiers for each of the above components even though viewing the different components as classifiers has not always been explicit [17]. The classifiers employed in state of the art SAT solvers have significantly improved over the years in their empirical performance, but there has been lack of rigorous analysis or theoretical understanding of the reasons behind the performance of these models. In this work, we focus on classifiers for *learned clause cleaning*, and we review two of the most prominent classifiers, employed in MiniSat and Maple_LCM_Dist. It is worth noting that MiniSat has been one of the most prolific SAT solvers, significantly faster than any other solver at the time of its release and Maple_LCM_Dist won the 2017 SAT Competition Main track.

The classifier of MiniSat. MiniSat maintains a limit on the maximum number of clauses in the memory, which is geometrically increased every time clause cleaning is performed. To this end, MiniSat keeps track of the *activity* of learned clauses. For a clause C, its *activity* is incremented every time C participates in the *1st Unique Implication Point (1st UIP)* conflict [4]. During clause cleaning, learned clauses are sorted in decreasing order by their *activity* and the bottom half of the learned clauses are thrown away.

The classifier of Maple_LCM_Dist [**14**]. This 2017 SAT Competition winning solver has a 3-tier system for keeping learned clauses: *Tier 0*, *Tier 1*, and *Tier 2*. *Tier 0* is never cleaned, i.e., clauses in *Tier 0* are never removed, while *Tier 1* is cleaned every 25K conflicts and *Tier 2* is cleaned every 10K conflicts. The different tiers' classifiers and the movement between the tiers is relatively complex, and we refer the interested reader to the Appendix for a detailed description of them.

2.1 Related Work

We assume that the reader is familiar with the SAT problem and for lack of space, we refer the reader to [4] for an extensive survey of related literature. While CrystalBall, to the best of our knowledge, is the first framework to provide white-box access to the execution of SAT solver; our work, nonetheless, makes use of several ideas from the extensive research pursued by the SAT community.

Xu et al. [25] proposed one of the earliest approaches to using machine learning for SAT solving. Their approach, SATZilla, focused on predicting the best SAT solver from a given portfolio of different solvers. SATZilla employed a supervised machine learning training process and focused on runtime as the metric. Recently, the SAT community has focused efforts to understand the performance of SAT solvers from different angles such as empirical studies focused on runtime [11,3,10], through the lens of proof complexity [7], and the like. In a series of papers, Liang et al. [13,16,15] have proposed usage of metrics other than runtime such as learning rate, global learning rate and the like [17]. Similarly, NeuroSAT [22] showcases the potential for ML in SAT, but does so using non-explainable deep learning with single-bit supervision.

3 CrystalBall: An Overview of the Framework

We now present the primary technical contribution of our work, CrystalBall focusing on designing the algorithmic ideas for keeping and throwing away learned clauses. Ideally, one would want to record the entire trace of the execution of the SAT solver for a given instance and perform classification on the collective traces on several instances. Given the complexity of modern SAT solvers, recording the entire trace is time and space consuming and cannot realistically scale beyond small SAT instances [23]. Since a learned clause can be viewed as derived by the application of propositional resolution, our insight is to employ DRAT to reconstruct a close approximation to the significant aspects of the trace of the SAT solver. Since we are using DRAT, our focus is limited to the execution of SAT solver on UNSAT instances. We designed the framework of CrystalBall to allow integration of most modern CDCL-based SAT solvers. For our work, we have integrated CrystalBall with CryptoMiniSat, a modern competitive SAT solver that was placed 3rd in the recently held SAT'18 competition.

CrystalBall consists of four phases: (i) feature engineering, (ii) data collection, (iii) data labeling, and (iv) classifier creation. *Feature engineering* focuses on the design of features that can be used by the classifier. *Data collection* focuses on the modifications to the SAT solver required for an efficient collection of reliable data and computation of labels corresponding to each learned clause. *Labeling* focuses on labeling the learned clauses whether to keep them or throw them away, based on total knowledge, i.e., past and future, of the learned clause *Classifier creation* focuses on employing state of the art supervised machine learning techniques to predict the label based on the past performance (i.e., data available while solving) of the learned clause.

The objective of CrystalBall is not to replace *expert intuition* but to allow for an *expert in the loop* paradigm where a significant amount of relevant data is made available to the expert to allow both for validation of ideas as well as to inspire new ones.

3.1 The Base Solver

As discussed in Section 2, the state of the art SAT solvers are comprised of multiple interdependent components, e.g., the learned clauses kept in memory influence clause learning, which influences the activity of variables and therefore branching and in turn, affecting the restart and clause deletion. The interdependence of the various components is both a challenge and an opportunity in the collection of data.

The complexity of interactions is a challenge, as it influences the data we gather in ways that are sometimes hard to understand. However, the interdependence of these components is what makes a SAT solver useful in solving real-world problems, and hence collecting this (sometimes messy) data makes the data useful. Collecting "clean" data would make little, if any, sense as the data would be useful neither to make inferences about what modern SAT solvers do nor to train a system to delete learned clauses in a modern SAT solver.

Our base solver to collect data exhibits the following set of dynamic behaviors, all of which are part of standard CryptoMiniSat, except for not explicitly deleting learned clauses:

1. We employ standard VSIDS variable branching heuristic as introduced in Chaff [19] as well as a learning rate based heuristic [16] along with polarity caching [21].
2. We use a mix of geometric [6] and Luby sequence [18] based static restart heuristics as well as a LBD score-based dynamic restart heuristic.
3. We perform inprocessing as standard for CryptoMiniSat. CryptoMiniSat does not perform preprocessing.
4. We strive to keep all learned clauses in memory since we want to know when every learned clause is useful in the unsatisfiability proof. Note that inprocessing may delete learned clauses in some cases.

3.2 Feature Engineering

The accuracy of typical supervised models is often correlated with presence of large number of features and large training data. This comes at the cost of training time and additional complexity of training process. Since training is an offline process and needs to be performed only once, we focus on design of a large number of features corresponding to learned clause. Our features can be categorized into four classes: (i) global features, (ii) contextual features, (iii) restart features, and (iv) performance features. We describe below different features in more detail with an intuitive rational for their inclusion.

Global features The global features of a learned clause are the property of the CNF formula at the time of clause creation. For example distribution statistics of horn clauses, irredundant clauses, number of variables, and the like. In particular, we use the features employed by SATZilla in the development of *portfolio solvers*. Since the underlying CNF formula undergoes substantial modifications during the solving, our solver recomputes these features regularly (in particular, at every 100K conflicts) unlike SATZilla that focuses on features only at startup. For every learned clause, we use the latest generated set of global features. Intuitively, inclusion of these features allows the classifier to avoid overfitting to particular types of CNF instances.

Contextual features To capture the context in which a clause is learned, we store features corresponding to the context of generation of a clause. In particular, contextual features are computed at the time of generation of the clause and relate to the generated clause. Contextual features include the number of literals in the clause or its LBD score.

Restart features Restarts constitute a core component of the modern SAT solvers and one can view every restart corresponding to a phase in the execution of a SAT solver. Intuitively, one expects restart to capture the state of the solver and the progress achievable from that state. We focus on features that capture the execution of the SAT solver in the current and preceding restarts. In particular, the restart features correspond to statistics (average and variance) on the size and LBD score of clauses, branch depth, trail depth during the current and previous restart.

Performance features The modern CDCL-based SAT solver maintain several performance parameters about learned clauses, which influence cleaning of the learned clauses. For example, as stated above, the classifiers in Maple_LCM_Dist employ *touched* and *activity* for ordering of clauses in Tier 1 and Tier 2 respectively. Activity is the clause activity as measured by MiniSat [6] and *touched* is the last conflict at which the system played part in a 1st UIP conflict clause generation. Consequently, we compute and maintain several performance features such as the number of times the solver played part of a 1st UIP conflict clause generation, the number of times it caused a conflict and the number of times it caused a propagation.

Normalization Ideally we want our features to be independent of the problem so that we can compare the values of a feature across problems. Our original features are the property of the particular run of the SAT solver. For different problems the absolute value of same feature can differ drastically. Therefore we can not directly compare the feature values across different problems. Instead, we have to rescale the feature values so that they become somewhat comparable across different problems. In order to achieve this we *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. This normalization is not perfect. However it does help in reducing the difference of scales of the the same feature across different problems. For instance the absolute learned clause size can vary drastically (10 vs 100) across different problems. However, the *relative learned clause size* feature becomes comparable across different problems

3.3 Data Collection

The data collection consists of two passes: a forward pass and a backward pass.

- **Forward pass**: The SAT solver is run on each of the benchmark formulae. During execution, we keep track of a set fraction of randomly chosen learned clauses, which we call *marked* learned clauses. For each *marked* learned clause C, we track and calculate its features and characteristics and continuously write them to a database file for later analysis. A DRAT proof is produced while running.
- **Backward pass**: A modified DRAT-trim [24] is used to parse the DRAT proof. This modified proof checker writes data into the same database file about each and every use of all *marked* learned clause in the unsatisfiability proof.

It is worth noting that while we keep track of learned clauses during the forward pass, we do not track how they were learned, i.e., which resolution rules were used to learn them. Tracking the resolution proof during the forward pass has been long thought to be computationally intractable for all except toy benchmarks. Recent works in progress have made some interesting headway; sustained development in this area may lead to versions of CrystalBall with the ability to keep track of proof in the forward trace. [4]

Our approach of using DRAT-trim has a number of advantages and disadvantages. The primary disadvantage arises from the fact that we are referring to two different proof trees during the forward and backward passes, i.e., proof tree generated by solver (which we are not tracking) might be different from the proof generated by DRAT-trim during the backward pass. However, this division of responsibility between the solver and the proof checker saves significantly on computational, and implementation efforts, and, most importantly allows the system to be used universally with minimal modification, as all competitive SAT solvers, and even many older ones, contain DRAT-trim support thanks to it being mandatory to participate in modern SAT Competitions.

Tracking and Sampling

We attach an ID, C_{ID}, to each clause C so that C can be correctly and fully tracked. We only track some randomly selected set of clauses due to size and timing constraints. All non-tracked clauses' C_{ID} is set to 0. We modified both the SAT solver to output and DRAT-trim to read and store, the 64-bit clause ID for each clause in the binary DRAT-trim format. We randomly set 96% of clauses' C_{ID} to 0 to have sufficient data without severe adverse effect on performance, thus being able to handle larger instances. Neither the forward nor the backward passes then wrote any data about clauses with C_{ID} of 0.

Forward Pass Data Gathering with SQLite

Dumping gigabytes of data for later analysis is a non-trivial task because it has to achieve simultaneously the convenience of data access, speed, and consistency

[4] Private communication: J. Nordström

of data collected across different runs. The data thus collected can significantly affect the quality of classifiers built on top. To dump the data, we chose SQLite because (a) it is self-contained, requiring no separate SQL server process, and (b) the data created by SQLite is a single file that can be efficiently collated with other files and copied from cloud and cluster systems, which is not the case for most of the other SQL servers' data files. Furthermore, SQLite is a mature, well-performing database with a complex query language, subqueries, indexes and the like.

By default, SQLite is synchronous. However, in our case, if execution fails, DRAT-trim also fails to generate data, and the run would be unusable in any case. The default SQLite choice of synchronicity comes at the expense of significantly increased runtime, with no benefits in our case. Therefore we set `PRAGMA synchronous = OFF` and `PRAGMA journal_mode = MEMORY` to increase the speed of writing data to disk substantially. Whenever possible, we also `INSERT` multiple data in one transaction, using SQLite's `BEGIN/END TRANSACTION` methods to lower synchronization overhead. All data dumping is done using *prepared queries*, creating precompiled queries for all data-dumping operations that later use raw C/C++ operations to write data. Hence, query strings are interpreted only once at the start, and the data is copied only once from the SAT solver's memory into SQLite library memory space, to be written to disk later.

To minimize overhead, we do not create indexes for any tables at table creation – instead, we create all needed indexes before querying. This eliminates the overhead of index maintenance during data collection. Indexes are also dropped and re-created when needed during data analysis for the same reason. Significant performance improvement can come from *not* having certain indexes, or more properly, having the right indexes only.

These usage details turn SQLite into a structured, fast, raw data-dumping solution that can later be used as a full-fledged SQL query system. This was key to a viable solution.

Backward Pass Data Gathering with DRAT-trim

Given a propositional formula φ and a clausal proof, DRAT-trim validates that the proof is a certificate of unsatisfiability of the formula φ. To this end, DRAT-trim first forward-searches the CNF, and the SAT-solver generated a proof file for the empty clause. Once it finds the empty clause, it runs through the proof file in a reverse fashion, recursively marking all clauses that contributed to the empty clause. To use DRAT-trim, we modified both the solver and DRAT-trim to write and, respectively, read, the conflict number $\mathsf{ConflNo}$ at which each clause is generated. DRAT-trim then knows for each learned clause what conflict number it is generated at. When verifying the proof, DRAT-trim uses this information to infer what conflict numbers each clause is used at. During DRAT-trim's backward pass, for all clauses $C_{ID} > 0$, the data pair of $C_{ID}, \mathsf{ConflNo}$ is dumped into a database $\mathsf{drat-data}$.

Recall, DRAT-trim does not have the information about the participation of the clauses in conflict generation during the forward pass of the solver. DRAT-trim can only infer that given the clauses in the database, the conflict could have taken place. It is possible that there are two sets of clauses, $\mathbb{A}$ and $\mathbb{B}$, $\mathbb{A} \neq \mathbb{B}$, both (potentially overlapping) sets are in the clause database of the solver and given either set, the conflict could have taken place. DRAT-trim employs a greedy algorithm to pick one of the two sets.

In general, it is possible to construct examples where an exponential number of clause sets could have caused a conflict and finding a minimum set may be neither trivial nor necessarily useful in building a classifier. It is, however, an exciting avenue of research to optimize the proof by making DRAT-trim take the smallest set at every point, for a particular post-processing overhead. This could allow training a classifier that could optimize for proof size. Exploring such extensions is beyond the scope of this work, and we leave it to future work.

3.4 Data Labeling

To infer a classifier via supervised learning, the inference engine requires the data corresponding to each clause be labeled a clause to be kept or thrown away. A naive strategy would be to label a clause *useful* if the clauses is used at all in the UNSAT proof generated by DRAT-trim. Analysis of data gathered from the backward pass indicates that proofs generated by CryptoMiniSat use close to 50% of the kept learned clauses while simultaneously throwing away approximately 99% of its learned clauses. This apparent contradiction is due to two factors: (1) as evidenced from the data gathered, most clauses are used in a hot spot close to where they are learned and not (or rarely) used later (2) SAT solver developers understand that there is a cost that is paid (both in memory usage and, more importantly, CPU clock cycles) for keeping a clause. Hence, if a clause is mostly useful in the near future, but may have, say, a single use far in the future, it may be beneficial to throw it away after a short amount of time, having served most of its purpose, and hoping that the solver will find a way to the proof anyway.

Given this analysis, it is clear that there are two corresponding guiding factors that govern whether we should label a clause to be kept: (1) whether the clause is useful at all later and (2) whether the distribution of future uses merits the solver to keep the clause, i.e. whether the cost should be paid to keep the clause until the future point(s) when it's useful. Satisfying requirement (1) is trivial when labeling a data point for keep/throw, given that we know when a clause will be useful and hence we know its last-use point. However, there are no existing cost models to satisfy (2). While a detailed study of construction of cost models is beyond the scope of this work and deferred to future work, we define the usefulness of a clause and the desired classifiers as follows:

1. A clause c is labeled to be kept for an interval of t conflicts if the number of times it was used in the final unsatisfiability proof (as computed by DRAT-trim) is greater than the average of the number of times all the clauses in databases are useful over the interval.
2. We employ two labelings (and associated classifiers) to handle the short-term and long-term usefulness of a clauses. The classifier *keep-short* (resp. *keep-long*) is short-sighted (resp. long-sighted) and attempts to predict whether the

clause is to be kept for the next 10K (resp. 100K) conflicts and is trained by using the data labeled by setting $t = 10,000$ (resp. $t = 100,000$) as explained above.

3.5 Inference of a Classifier via Supervised Learning

Given the labeled data, we considered several choices for our classifier including SVM, decision trees, random forests, and logistic regression. The desired learning algorithm was chosen based on the following primary constraints: (1) Our 218 features, comprised of four different categories, are mixed and heterogeneous, (2) there is no straightforward way to normalize all of our features, (3) the model must be easily convertible to C++ code as the decision inside the SAT solver has to very fast, and (4) the model must provide meaningfully good prediction accuracy.

Although we have created good classifiers using both SVM and logistic regression, we found that satisfying requirement (3) is relatively complicated for these and tuning them in our chosen framework, scikit-learn [20], is harder than decision trees and random forests. Decision trees satisfy all requirements and allow for easy visualization but give relatively worse prediction accuracy than random forests. We therefore chose random forests as the classifier for our classifier when running in the solver, and decisions trees when visualizing and debugging the decision logic during training.

For demonstration purposes, both a *keep-short* and a *keep-long* trained decision tree is visualized in the Appendix. As expected, the actual random forests used are too big to be visualized, containing approx. 320 decision nodes for each classifier's tree, where 10 trees make up a decision forest. The prediction accuracy of the decision trees visualized are only slightly lower than the final decision forests, and reviewing them can lead to interesting insights.

3.6 Feature Ranking

As discussed in Section 2, state of the art solvers over the past decade have relied on finding and exploiting strong features to estimate learned clause quality. These relatively few features form the core of their heuristic for learned clauses database management. The identification of these heuristics has largely been driven by expert intuition and on runtime measurements.

In contrast, CrystalBall employs a data-driven heuristic along with an expert-driven cost model, for identifying the distinguishing features, which we can chose to be a relatively large set, given that the classifier will be inferred automatically and complicated relationships between them will be handled by the classifier. We use feature importance method of the Random Forest to rank the features by their importance. A Random Forest consists of several decision trees. The root of a decision tree corresponds to the most important decision in the decision tree as the root affects the classification for a large fraction of inputs. Furthermore, the decision nodes closer to the root are more important while the ones farther away from the root are less important. Quantitatively CART 4.5 uses the *average* *decrease in Gini impurity* as the measure of feature importance. Gini impurity is a measure of how often a randomly chosen element from the set would be incorrectly labeled if it was randomly labeled according to the distribution of the labels in the subset. The importance of a feature in a Random Forest is the average importance of the features in each decision tree. The importance of a feature in a decision tree is the decrease in Gini impurity caused by decisions involving that feature in the decision tree [9].

4 Results

As stated in the introduction, the mission of CrystalBall is to develop a framework to provide white-box access to the execution of SAT solver. The current version of CrystalBall aims to understand the algorithmic ideas for keeping and throwing away learned clauses.

To conduct experiments, we used a high-performance computer cluster, where each node has an E5-2690 v3 CPU with 24 cores and 96GB of RAM. We used all the 934 unique CNFs from SAT Competitions' 2014,'16 and '17 both to obtain training data then to evaluate the speed of the final solver executable — however, following standard practice, we split the data into 70% for training and 30% for testing and also did not (and could not) use satisfiable instances for training, which constituted about 45% of all instances. The experimental results presented in this paper required over 250,000 CPU hours (equivalent to 28 CPU years)

Training Phase. When collecting data, we used a 12h timeout for both the solver and DRAT-trim and generated over 37 GB of SQLite data. Since the number of clauses learned for different problems varied widely, we sampled a fixed set of data points from each benchmark to ensure fair representation and discarded problems that were solved too fast to be meaningful. After sampling, we have $\approx 429K$ data points for the *keep-short* classifier and $\approx 85K$ data points for the *keep-long* classifier. Each data point contained the 200+ features plus the label to keep or throw away the clause.

Testing Phase. While solving, we used a 5000s timeout and 4GB memory limit, which is in line with general SAT Competition rules. Recall, we used two classifiers *keep-short* and *keep-long*. If either of the two classifiers triggers, the clause is kept. However, if the *long-keep* triggers, the clause will not be deleted for the next 100K conflicts.

4.1 Accuracy of the Classifiers

As described in Section 3.4, we train two binary classifiers whether to keep or throw away a clause at every $N = 10,000$ conflict ticks. The confusion matrix for the classifiers *keep-short* and *keep-long* for the test data are shown in Table 1. A careful design of the objective function is mandated by the solver's widely different actions corresponding to the predicted label. In case one incorrectly predicts that a clause should be thrown away at tick k and then would correctly predict to keep it at tick $k + 1$, it is already thrown away and the system has

		Prediction	
		Throw	Keep
Ground truth	Throw	0.64	0.36
	Keep	0.11	0.89

(a) *keep-short*

		Prediction	
		Throw	Keep
Ground truth	Throw	0.63	0.37
	Keep	0.09	0.91

(b) *keep-long*

Table 1: Confusion matrix for the trained classifiers

already failed. Hence, it is important to try to err on the side of caution and sway our classifiers towards keeping clauses. Hence, we gave twice the sample weight to predicting to keep a clause relative to throwing it away. This focuses on minimizing the error cases where the classifier incorrectly predicts that the clause should be thrown away, the false negative scenario. Therefore the false positive and false negative error rates are not symmetric for our case, ≈ 0.35 vs. ≈ 0.10. All in all, the high accuracy of cells in the confusion matrix shows the potential of the data-driven approach for classification of whether to keep a clause or not.

4.2 Insights from Feature Ranking

Using the feature ranking method through a random forest classifier (as describe above), we obtain the feature rankings and importance scores present in Table 2. Note that since we train two classifiers, with two different labelings, we obtain two rankings and two set of importance scores. In fact, the two differ in ways that are quite interesting. The *keep-short* classifier identifies the well-known "last used in a 1st UIP conflict" as the most important feature. On the other hand, the *keep-long* classifier identifies the total number of times a learned clause participated in a 1st UIP conflict as the most important feature. A generally interesting observation is that all features identified in the top 10 for both classifiers are dynamic features i.e., the said features are not computed at clause creation such LDB (Literal Block Distance), used by most solvers. A possible explanation would be that features such as LBD may be most useful for a classifier that would predict to keep the clause forever (what one could call *keep-forever*), instead of re-examining the clause every N conflicts. We defer the design of such a classifier to future work.

4.3 Solving SAT Competition CNFs

Given the insights from the feature ranking, it is crucial to perform a performance analysis of the learned model for validity and more in-depth insight. The straightforward method is to augment the base solver, CryptoMiniSat v5.6.8, with the model inferred by CrystalBall. To this end, we have performed a preliminary study by using 22 features selected using the guidance given by the top feature list, as marked in Table 2. Implementation and testing of classifiers based on a more extensive set of features is beyond the scope of this study and left for

Feature	Relative importance
rdb0.used_for_uip_creation*	.1121
rdb0.last_touched_diff*	.1052
rdb0.activity_rel	.0813
rdb0.sum_uip1_used**	.0635
rdb1.sum_uip1_used**	.0631
rdb1.activity_rel	.0521
rdb1.last_touched_diff*	.0486
rdb1.act_ranking_top_10*	.0457
rdb0.act_ranking	.0442
rdb0.act_ranking_top_10*	.0416
rdb1.act_ranking	.0403
rdb0.sum_delta_confl_uip1_used**	.0304
rdb1.used_for_uip_creation	.0296
cl.antecedents_lbd_long_reds_var*	.0189
rdb1.sum_delta_confl_uip1_used	.0183
cl.lbd_rel*	.0172
rdb.rel_last_touched_diff*	.0162
cl.lbd_rel_queue*	.0138
rdb.rel_used_for_uip_creation*	.0135
cl.lbd_rel_long*	.0126

(a) Best features for *keep-short*

Feature	Relative importance
rdb0.sum_uip1_used*	.1304
rdb1.sum_uip1_used*	.0983
rdb0.used_for_uip_creation*	.0774
rdb0.act_ranking	.0740
rdb0.act_ranking_top_10*	.0511
rdb0.last_touched_diff*	.0489
rdb1.act_ranking	.0481
rdb0.sum_delta_confl_uip1_used*	.0435
rdb1.used_for_uip_creation	.0435
rdb0.activity_rel	.0416
rdb1.act_ranking_top_10*	.0351
rdb1.last_touched_diff*	.0346
rdb1.sum_delta_confl_uip1_used	.0293
cl.lbd_rel*	.0217
cl.lbd_rel_queue*	.0176
cl.size_rel*	.0152
cl.size*	.0148
cl.antecedents_lbd_long_reds_max	.0146
rdb1.activity_rel	.0139
cl.lbd_rel_long*	.0135

(b) Best features for *keep-long*

Table 2: Table of feature rankings. We refer the reader to Appendix for interpretation of each of the features. Features marked with a $*$ were used in both classifiers, except for the ones marked with $**$ that were left out of the *keep-short* classifier. The only feature not present in these rankings but used is dump_no, the number of times a learned clause has been up for deletion.

future work. We compare the augmented solver vis-a-vis Maple_LCM_Dist, whose tiered set of classifiers serves as inspiration for our classifiers *short-keep* and *long-keep*. We performed the comparison over all the unique 934 instances from SAT Competitions 2014-17 with a timeout of 5000 seconds.

In summary, CryptoMiniSat augmented with our classifiers, referred to as PredCryptoMiniSat, could solve 612 formulas, obtaining a PAR-2 score[5] of 3761077 while Maple_LCM_Dist could only solve 591 obtaining a PAR-2 score of 4039152. It is worth recalling that our classifiers are learned only using UNSAT instances and therefore, we had data corresponding to only 236 out of 945 formulas. In particular, PredCryptoMiniSat solved 271 satisfiable instances and 341 unsatisfiable instances,

[5] PAR-2 score is defined as the sum of all runtimes for solved instances + 2*timeout for unsolved instances, lowest score wins. This scoring mechanism has been used in most recent SAT Competitions

which is in line with the distribution of known SAT and UNSAT instances among the problems. The ability of the learned classifier to handle SAT instances almost as well as UNSAT instances, along with being able to outperform the state-of-the-art solver of 2017, provides strong evidence in support for the component design choices of CrystalBall.

It is essential to analyze the above results through an appropriate lens. First, the latest hand tuned model in CryptoMiniSat allows it to solve 637 formulas with a PAR-2 score of 3506488. Although this is significantly better than PredCryptoMiniSat, it has been tuned over many years. Secondly, our model is not optimized for memory consumption which leads to significantly increased cache misses, hence increased runtime. To be able to use all 22 features, PredCryptoMiniSat keeps an additional 68 bytes of data for each clause. Furthermore, a fine-tuned version with fewer features and perhaps more than two classifiers is likely to result in improved runtime. Therefore, overall we believe that our learned model not only highlights surprising power of several features but could also be a starting point to design state of the art solvers by using auto-generated data-driven yet interpretable models.

5 Conclusion

In this paper, we introduced to the SAT community our framework, CrystalBall, to analyze and generate classifiers using significant amounts of behavioral data collected from the run of a SAT solver. CrystalBall combines data collection of its forward pass with proof data using DRAT-trim in its backward pass to allow studying more than 260 UNSAT instances from SAT Competitions. Our preliminary results demonstrate the potential of data-driven approach to accurately predict whether to keep or throw away learned clauses and to rank features that are useful and in this prediction. Our experiments were able to not only derive interesting set of features but also demonstrate the strength of our solver on competition benchmarks.

As a next step, we plan to extend CrystalBall to allow easier integration with other state of the art SAT solvers. In the long term, we believe CrystalBall will both enable to better understand SAT solvers and lower the barrier to designing heuristics for high-performance SAT solvers. Finally, given the requirement of tight integration of learned models into state of the art SAT solvers, CrystalBall presents exciting opportunities for the design of interpretable machine learning models.

Acknowledgements This research is supported in part by the National Research Foundation Singapore under its AI Singapore Programme AISG-RP-2018-005 and NUS ODPRT Grant R-252-000-685-133. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore https://www.nscc.sg.

References

1. Proceedings of SAT Competition 2018; solver and benchmark descriptions (2018)
2. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) IJCAI. pp. 399–404 (2009)
3. Biere, A., Fröhlich, A.: Evaluating cdcl variable scoring schemes. In: Proc. of SAT. pp. 405–422 (2015)
4. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability. IOS Press (2009)
5. Davis, M., Putnam, H., Robinson, J.: The decision problem for exponential diophantine equations. Annals of Mathematics pp. 425–436 (1961)
6. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Theory and Applications of Satisfiability Testing, SAT 2003. Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer (2003). https://doi.org/10.1007/978-3-540-24605-3_37, https://doi.org/10.1007/978-3-540-24605-3_37
7. Elffers, J., Giráldez-Cru, J., Gocht, S., Nordström, J., Simon, L.: Seeking practical CDCL insights from theoretical SAT benchmarks. In: Proc. of IJCAI. pp. 1300–1308 (2018)
8. Elffers, J., Johannsen, J., Lauria, M., Magnard, T., Nordström, J., Vinyals, M.: Trade-offs between time and memory in a tighter model of CDCL SAT solvers. In: Proc. of SAT. pp. 160–176 (2016)
9. Friedman, J., Hastie, T., Tibshirani, R.: The elements of statistical learning, vol. 1. Springer series in statistics New York, NY, USA: (2001)
10. Jamali, S., Mitchell, D.: Centrality-based improvements to cdcl heuristics. In: Proc. of SAT. pp. 122–131 (2018)
11. Katebi, H., Sakallah, K.A., Silva, J.P.M.: Empirical study of the anatomy of modern SAT solvers. In: Proc. of SAT. pp. 343–356 (2011)
12. Lederman, G., Rabe, M.N., Seshia, S.A.: Learning heuristics for automated reasoning through deep reinforcement learning. Proc of. ICLR (2019)
13. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Exponential recency weighted average branching heuristic for SAT solvers. In: Proc. of AAAI. pp. 3434–3440 (2016)
14. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning Rate Based Branching Heuristic for SAT Solvers. In: Proc. of SAT (2016)
15. Liang, J.H., Oh, C., Mathew, M., Thomas, C., Li, C., Ganesh, V.: Machine learning-based restart policy for cdcl sat solvers. In: Proc. of SAT. pp. 94–110. Springer (2018)
16. Liang, J.H., V.K., H.G., Poupart, P., Czarnecki, K., Ganesh, V.: An empirical study of branching heuristics through the lens of global learning rate. In: Proc. of SAT. pp. 119–135 (2017)
17. Liang, J.: Machine learning for SAT solvers. Tech. rep., University of Waterloo (2018), PhD Thesis
18. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. Inf. Process. Lett. **47**(4), 173–180 (1993)
19. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: Proc. of DAC. pp. 530–535. ACM (2001)
20. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)

21. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: SAT. pp. 294–299 (2007)
22. Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., Dill, D.L.: Learning a SAT solver from single-bit supervision. Proc of. ICLR (2019)
23. Shacham, O., Yorav, K.: On-the-fly resolve trace minimization. In: Proceedings of DAC. pp. 594–599. ACM (2007)
24. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) Proc. of SAT. pp. 422–429 (2014)
25. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT **32**, 565–606 (06 2008)

A Appendix

A.1 Tiers in Maple_LCM_Dist

The following are the learned clause tiers in Maple_LCM_Dist with their associated classifiers for keeping/throwing away a clause, and for tier-movement logic.

Tier 0 Clauses with *LBD score* lower or equal to 4 is placed into *Tier 0*.[6] In case the solver has ran for more than 100k conflicts and *Tier 0* contains less than 100 clauses, then the solver puts all the clauses with LBD score smaller or equal to 6 into *Tier 0* for rest of the execution.

Tier 1 Every learned clauses not put in Tier 0 is put in Tier 1 when the clause is learned. *Tier 1* is cleaned every 25K conflicts. During cleaning process, if a clause C has not been used in the past 30K conflicts during 1st UIP and if the clause was learned more than 30K conflicts ago then the clause is moved to *Tier 2* and its activity is reset to 0 and bumped once.

Tier 2 A learned clause in Tier 2 is cleaned every 10K conflicts by sorting in decreasing order the clauses according to their *activity* and deleting the bottom half of them. The *activity* of the clause is calculated as per MiniSat [6], i.e., the activities of clauses are bumped when they are used during UIP and their activities are exponentially decreased (relative to all other clauses).

A.2 An Example Prediction Tree

In Fig. 1 is an example decision tree for the *keep-short* classifier. This tree is significantly cut down by asking the system to generate a tree where a split happens only when at least 3% of data is present at a leaf. Note that since we had over 400k data points, 3% of data in our case is over 10k data points. To improve accuracy, one should make a split at a much lower percentage point.

[6] Note that core_lbd_cut in the source is set to 3, however, the LBD score is decremented by one before the check happens.

Table 3: Description of the features in Table 2

Feature	Description
cl.antecedents_lbd_long_reds_max	Clause's antecedents' if they are non-binary and learned, their maximum LBD value
cl.antecedents_lbd_long_reds_var	Clause's antecedents' if they are non-binary and learned, the variance of their LBD value
cl.lbd_rel	LBD of the clause, relative to the LBD of all previous learned clauses in the restart
cl.lbd_rel_long	LBD of the clause, relative to the LBD of all previous learned clauses
cl.lbd_rel_queue	LBD of the clause, relative to the LBD of at most the past 50 previous learned clauses in the restart
cl.size	Number of literals in the clause
cl.size_rel	Size of the clause, relative to the size of all previous learned clauses.
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.
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.
rdb0.act_ranking_top_10	Whether the activity of the clause belongs to the top 10%, top 20%, ..., 100%. among all learned clauses at the point of time when the decision to keep or throw away the clause is made.
rdb0.last_touched_diff	Number of conflicts ago that the clause was used during a 1st UIP conflict clause generation.
rdb0.sum_delta_confl_uip1_used	rdb0.sum_uip1_used divided by rdb0.sum_delta_confl_uip1_used
rdb0.sum_uip1_used	Number of times that the conflict took part in a 1st UIP conflict generation since its creation.
rdb0.used_for_uip_creation	Number of times that the conflict took part in a 1st UIP conflict generation since its creation.
rdb1.activity_rel	Same as rdb0.activity_rel but instead of the current round, it is data from the previous round (i.e. 10k conflicts earlier)
rdb1.act_ranking	Same as rdb0.act_ranking but instead of the current round, it is data from the previous round (i.e. 10k conflicts earlier)
rdb1.act_ranking_top_10	Same as rdb0.act_ranking_top_10 but instead of the current round, it is data from the previous round (i.e. 10k conflicts earlier)
rdb1.last_touched_diff	Same as rdb0.last_touched_diff but instead of the current round, it is data from the previous round (i.e. 10k conflicts earlier)
rdb1.sum_delta_confl_uip1_used	Same as rdb0.sum_delta_confl_uip1_used but instead of the current round, it is data from the previous round (i.e. 10k conflicts earlier)
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)
rdb.rel_last_touched_diff	Whether rdb0.last_touched_difffor rdb1.last_touched_diffis larger
rdb.rel_used_for_uip_creation	Whether rdb0.used_for_uip_creation or rdb1.used_for_uip_creation is larger
dump_no	Number of times this clause has been up for deletion already

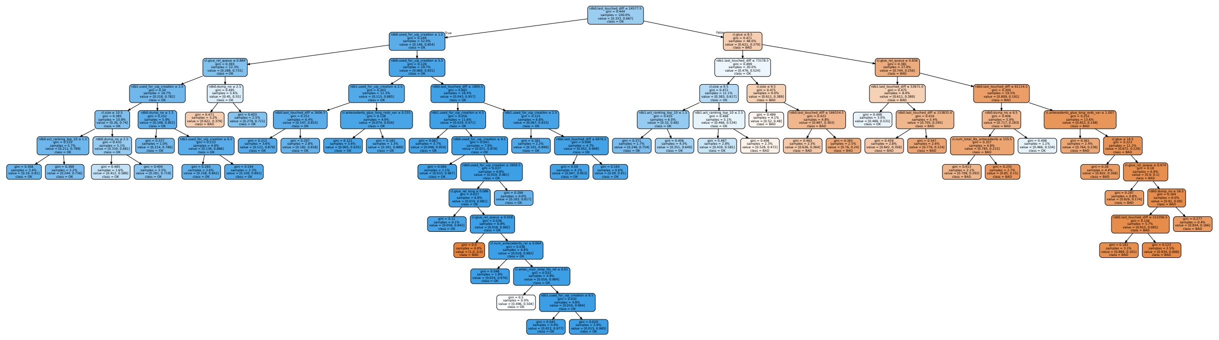

Fig. 1: Example decision tree for the *keep-short* classifier, where a case split was set to be performed only in if at least 3% of data is present. This greatly reduces the size of the tree. When the label “OK” is predicted, the clause is kept. When the label “BAD” is predicted, the clause is thrown away.

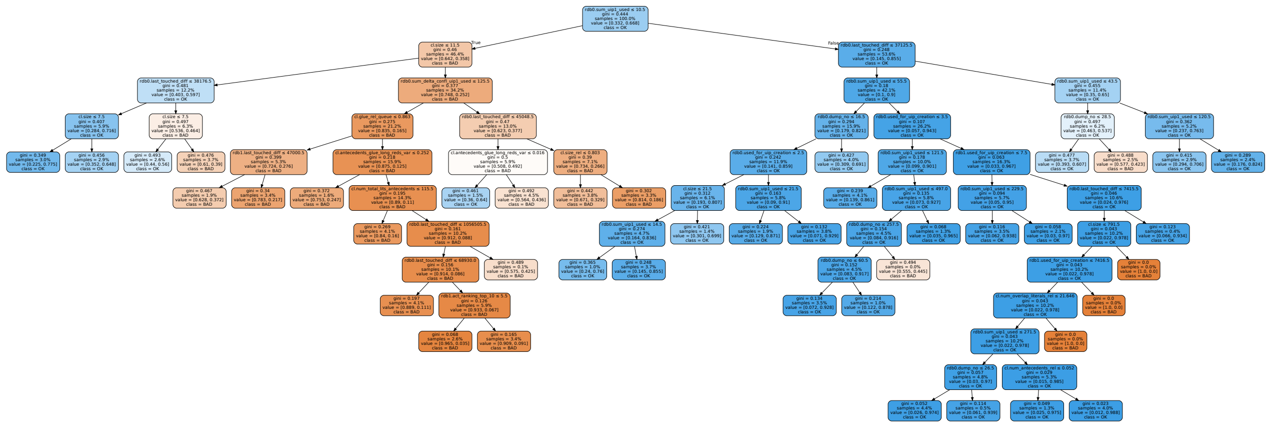

Fig. 2: Example decision tree for the *keep-long* classifier, where a case split was set to be performed only in if at least 3% of data is present. This greatly reduces the size of the tree. When the label "OK" is predicted, the clause is kept. When the label "BAD" is predicted, the clause is thrown away.