Towards Learning Quantifier Instantiation in SMT

Mikoláš Janota
Czech Technical University in Prague, Czechia

Jelle Piepenbrock
Czech Technical University in Prague, Czechia & Radboud University Nijmegen, Netherlands

Bartosz Piotrowski
Czech Technical University in Prague, Czechia & University of Warsaw, Poland

Abstract

This paper applies machine learning (ML) to solve quantified satisfiability modulo theories (SMT) problems more efficiently. The motivating idea is that the solver should learn from already solved formulas to solve new ones. This is especially relevant in classes of similar formulas.

We focus on the enumerative instantiation—a well-established approach to solving quantified formulas anchored in the Herbrand's theorem. The task is to select the right ground terms to be instantiated. In ML parlance, this means learning to rank ground terms. We devise a series of features of the considered terms and train on them using boosted decision trees. In particular, we integrate the LightGBM library into the SMT solver cvc5. The experimental results demonstrate that the ML-guided solver enables us to solve more formulas than the base solver and reduce the number of quantifier instantiations. We also do an ablation study on the features used in the machine learning component, showing the contributions of the various additions.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases satisfiability modulo theories, quantifier instantiation, machine learning

Digital Object Identifier 10.4230/LIPIcs.SAT.2022.7

Funding The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project *POSTMAN* no. LL1902, the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/0000466, the grant of National Science Center, Poland, no. 2018/29/N/ST6/02903, and Amazon Research Awards. This article is part of the *RICAIP* project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 857306.

1 Introduction

Solving formulas containing quantifiers in the context of Satisfiability Modulo Theories (SMT) is famously difficult. This difficulty is inherent since quantifiers lead to undecidability or high computational complexity [15, 23]. Nevertheless, quantifiers are indispensable in practical problems. Notably, in software verification, they are used to express properties of memory, e.g., that an array is sorted. This paper tackles the question: *Can machine learning (ML) make SMT solvers more efficient in the context of quantifiers?*

The potential of ML is to enable the solver to learn from problem instances that it has already solved. In contrast, current SMT solvers only take into account one formula during solving. However, integrating ML into this context is not straightforward. We are facing two main challenges:

1. ML operates in an approximate setting, while SMT is anchored in a rigorous background where inference steps need to follow the logic in question. This means the solver inference steps must be followed and the ML integrated into the solver framework.
2. SMT solvers make millions of decisions for a single problem. How can ML be integrated without dramatically slowing down the solver?

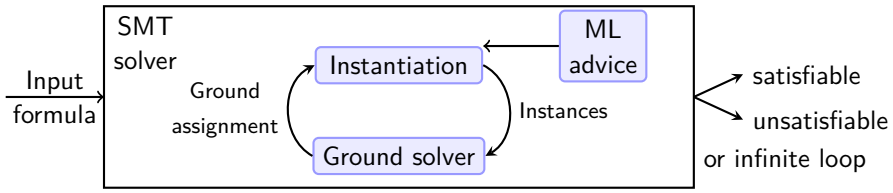

Figure 1 Schematic of the SMT solver with machine learning guidance for quantifier instantiation.

This paper proposes a design that enables ML to steer the state-of-the-art SMT solver cvc5 [2][1] in quantifier instantiation (see Figure 1).

A general technique to handle quantifiers in SMT is to gradually instantiate the quantified sub-formulas with ground terms until obtaining contradiction. For instance the formula $(\forall x\, f(x) > x) \wedge (\forall y\, f(y) < 0)$ is readily refuted by instantiating both x and y with 0.

The terms to be used in instantiations may be chosen either by making use of syntactic properties, e.g. by *e-matching* [11], or by utilizing semantic properties, e.g. *model-based quantifier instantiation* [16]. Interestingly, the complexity of these techniques may not always pay off: simple *enumerative instantiation* of terms can often give better results [36, 19]. For all of these techniques, the most important challenge is a large number of possible terms that can be chosen for instantiation, especially in later stages of solving.

In recent years, ML has been applied in countless settings, from computer vision [22] to natural language processing [12]. There is also work to learn decision-making for first-order theorem proving, cf. [18], but the use of ML for SMT guidance still has large untapped potential.

In this work, we use machine learning to improve the performance of cvc5 in real-time, by learning a scoring function for terms that guides the quantifier instantiation process. This addresses our first challenge, i.e., how to use an inherently approximate method within SMT. The second challenge of avoiding dramatic slowdown by ML within the solver is achieved by the choice of features, ML model, and its tight integration into the solver.

This paper has the following primary contributions.

1. We design an integration of ML guidance for quantifier instantiation in the context of SMT. In particular, the enumerative instantiation is guided by ML during the run of the solver, while learning from existing solutions to already solved problems.
2. We implement the proposed method in cvc5 and the implementation shows a significant increase in the number of solved instances and lowers the number of instantiations needed for many proofs.

2 Background

Throughout the paper we assume familiarity with first-order logic, in particular, with satisfiability modulo theories (SMT) [6]. Formulas with no quantifiers, called *ground formulas*, are solved using the DPLL(T) paradigm [29]. DPLL(T) abstracts first-order logic atoms as propositional variables enabling the use of a SAT solver to reason about the Boolean structure of the formula and *theory solvers* to reason about theories.

SMT solvers reason about quantifiers by instantiating with ground terms to strengthen the ground part of the formula. Effectively, a quantified sub-formula or quantified expression

[1] cvc5 is a successor to CVC4 [4, 2].

$(\forall x_1 \dots x_n\, \phi)$ is a source of lemmas of the form $(\forall x_1 \dots x_n\, \phi) \Rightarrow \phi\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$, where ϕ is quantifier-free and t_i are ground terms. For instance, $\forall x\, f(x) > 0$ may be instantiated as $(\forall x\, f(x) > 0) \Rightarrow f(0) > 0$. For simplicity, assume that quantifiers are removed on preprocessing by Skolemization.

This approach results in a loop that moves back and forth between a *ground solver* and the *instantiation module*; see Figure 1. The ground solver only sees quantifiers as propositions that either should hold or not. Once the ground solver finds a satisfying assignment for the ground part of the formula, control is handed over to the instantiation module, which generates new instances of the quantified sub-formulas that currently should hold. This strengthens the ground part of the formula and the process repeats. *Our contribution is to provide ML advice for the instantiation module with the aim of suggesting instantiations that lead to unsatisfiability in the ground solver.*

There is a bevy of methods for choosing instantiations. For decidable fragments, dedicated approaches exist, e.g., for bit-vectors or linear arithmetic [37, 13, 26, 7]. General quantifiers are most notably tackled by *e-matching*, based on syntactic properties of the terms [11] and *model-based* [16] or *conflict-based* [38] instantiation, relying on the semantics of the formula. Niemetz et al apply syntax-guided instantiation term generation [27].

The instantiation method we focus on here is *enumerative instantiation*. While the method is probably the most straightforward one, it has good performance on many SMT problem categories and adds to the robustness of the solver [36].

2.1 Enumerative Instantiation

Herbrand's theorem [17] guarantees that for an unsatisfiable first-order logic formula, finitely many instantiations are sufficient to obtain an unsatisfiable ground part, and, these instantiations only need to use the Herbrand universe. Completeness is not guaranteed in theories (e.g. $\forall x : \mathbb{R}\; x^2 \neq 2$). However, the application of the theorem in SMT is justified as it provides a viable way to deal with the complex problem of quantifier instantiation.

Reynolds et al. invoke a stronger variant of Herbrand's theorem that enables a more practical method for quantifier instantiation [36]. It is sufficient to consider only the terms already within the ground part of the formula generated so far. This insight leads to the *enumerative instantiation* strategy, the technique we augment with machine learning guidance in this work. For a formula $G \wedge \forall x_1 \dots x_n\, \phi$, with G ground, collect all ground terms $\mathcal{T}$ in G and strengthen G by an instantiation of ϕ by an n-tuple $t_1, \dots, t_n$ with $t_i \in \mathcal{T}$; repeat the process until G becomes unsatisfiable or until resources are exhausted. The tuples are enumerated systematically to guarantee fairness.

As a motivational (toy) example consider the following conjunctive set of formulas within the logic of uninterpreted functions and linear integer arithmetic (UFLIA).

$$\{f(d) > f(d+2),\; c \leq 0 \vee \underbrace{\forall x\, f(x) < f(x+1)}_{q}\}$$

For the ground solver, the quantifier is abstracted as a Boolean constant q and the instantiation module is responsible for adding lemmas of the form $q \Rightarrow f(t) < f(t+1)$ for some ground term t. Consider a context where the solver decides that $\neg(c \leq 0)$, which forces q to be true. Ideally, in this situation the solver instantiates x first with d and then with $d+1$, resulting

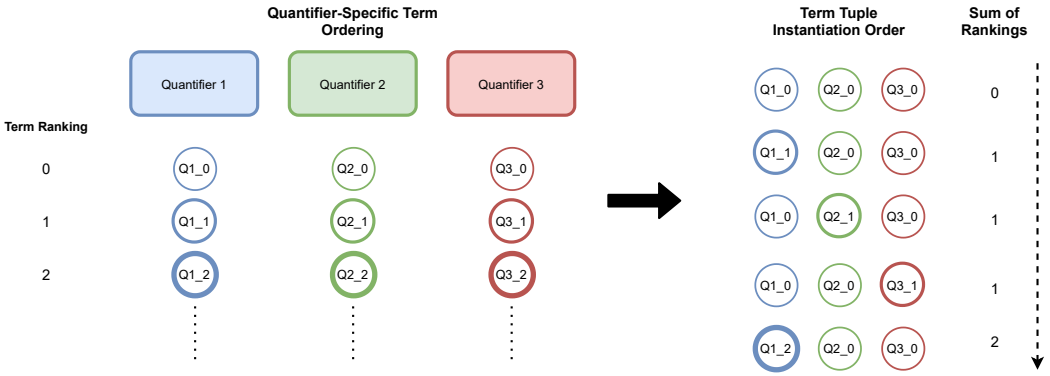

Figure 2 Schematic example of the term enumeration process with term tuple ordering based on the sum of term rankings. The task of the ML model is to take features based on the quantified formula, the variable, the terms, and the context to score the terms, thus changing the term ranking to deliver better instantiations. In cases where multiple quantifiers have to be instantiated at the same time, we instantiate with term tuples, which are ordered by the sum of the rankings of the individual terms.

in the following steps:

ground formula	**additional ground terms**
$\{c \leq 0 \vee q, f(d) > f(d+2)\}$	$\{c, 0, d, d+2, f(d), f(d+2)\}$
$\{q \Rightarrow f(d) < f(d+1)\}$	$\{d+1, f(d+1)\}$
$\{q \Rightarrow f(d+1) < f(d+2)\}$	$\{d+2, f(d+2)\}$

From transitivity of $>$, the ground part gives a contradiction for the current context, forcing q to false and $c \leq 0$ to true. Already this small example shows the difficulties we are facing. For instance, instantiating with the term $f(d+2)$ results in $q \Rightarrow f(f(d+2)) < f(f(d+2)+1)$, which not only is unhelpful but also produces a harder ground instance.

Individual instantiations lead to the addition of new ground terms into a sequence. We refer to the position of a term in the sequence as its *age*; in the above example, the term d has age 0. Terms added by the same instantiation are in the same *phase*; in the above example, the terms $d+1, f(d+1)$ are added in phase 2.

As an additional filter, cvc5 uses a technique called *relevant domain*, introduced by Ge and de Moura [16]. Intuitively, a term becomes relevant for a certain quantified variable if it appears in the same position as the variable, e.g., if $f(x)$ appears in the quantified formula and there is a ground term $f(t)$, the sub-term t is relevant for x; this is further closed by equality. By default, cvc5 first considers only instantiations by terms from the relevant domain and only after that moves on to the rest. Formulas with multiple quantified sub-formulas are solved by instantiating the sub-formulas independently but in a fair manner.

3 Learning Ordering of Terms

When using the enumerative instantiation strategy, the SMT solver uses an ordering of the available terms (by default, this is primarily based on the *age* of the terms) and enumerates terms in this order, trying instantiations. As shown in Figure 2, when there are multiple quantifiers that need to be instantiated with a tuple of terms, the solver uses the sum of the rankings of the individual terms to determine the ranking of the term tuple [19].

Cvc5 considers one quantified expression at a time. In this work, we also only consider the reranking of terms for a single variable at a time.[2] Thus, at each decision point of the quantifier module, we consider the current quantified expression Q, the variable V, the term T, and the wider context *Context*, containing all other quantified expressions and the ground part of the problem. Featurization is then viewed as a function $F : (Q, V, T, \mathit{Context}) \to \mathbb{R}^n$.

The machine learning heuristic's task is to reorder the term candidate lists for each quantifier so that more useful instantiations are tried earlier (see Figure 2). For this, we use a *scoring function* and rank the candidates according to this score. This scoring function $S \colon \mathcal{F} \to [0, 1]$ takes as its input the *features* obtained by applying the featurization F to a tuple $(Q, V, T, \mathit{Context})$.

The returned score is intended to reflect how likely it is that term T was used to instantiate variable V in quantified expression Q with the context *Context* in the final proof. The training data for the ML model is based on previously found proofs, so that we can extract a label for each tuple $(Q, V, T, \mathit{Context})$, based on what decision was made in a known proof of the problem. This label is 1 if the instantiation that the tuple represents was used in a successful proof (which we will call a *positive* example) or 0 otherwise. During solving, the scoring function S is applied to each candidate term. The terms are then sorted according to their score by *stable* sort, i.e., equally scored terms remain in their original order (see Figure 2).

The scoring function S is implemented as an ML-model. Specifically, we experimented with a logistic regression model and with gradient boosted decision trees (GBDT). In contrast to popular neural network methods, these methods are sufficiently fast to run at solving time within the solver loop [40]. In the initial experiments, the boosted trees performed significantly better than logistic regression, thus we keep it for the rest of our experiments.

GBDT uses an ensemble of decision trees, where decisions of all the individual trees are aggregated into a more reliable decision. Gradient boosted trees are widely used in machine learning applications. In particular, this algorithm is one of the most successful approaches in machine learning competitions [30]. They have also been used for machine learning guidance in first-order logic [18, 33]. After training, we use this ensemble of decision trees to predict the label of each candidate term. Ideally, the trees predict 1 if the candidate leads to a proof and 0 if it does not. In practice, the prediction of the model is a float number between 0 and 1, which can be interpreted as a probability. We used the library LightGBM [21], a fast and efficient implementation of the GBDT algorithm.

3.1 Featurization

The GBDT algorithm makes decisions based on a representation of the state of the solver, the relevant quantified expression and the term that is being considered. This representation of the data is called the *featurization* of the data. There are several categories of features with different properties. First, we list them here, afterwards, there are subsections with the details of the features in that category. Note that we use the single-letter abbreviations following the categories to denote them in figures in later sections. The designed features provide a simple characterization of the training examples. However, they are extracted using existing, efficiently implemented mechanisms of cvc5, which makes the process fast enough. All these design decisions enable us to perform ML prediction online, during the proof search. In Section 4.3.3, we show an ablation study, where impacts of each feature category for the performance of the solver can be observed.

[2] The enumeration still creates tuples of terms to instantiate quantified expressions with multiple variables.

Table 1 Short description of feature categories. The abbreviations are later used to concisely refer to these feature categories.

Category Name	Description	Abbreviation
Procedural Features	Data from the solver, such as the age of terms, number of times a term was tried.	p
Bag-of-words Features	Amounts of nodes of certain type in quantified expression or term.	b
Context Features	The parent symbols of variables and the parents of the head symbols in candidate terms.	c
Numeral Features	The minimum and maximum number used in quantified expression or term.	n
Parent Features (Parent Label Propagation)	Terms that are necessary to create the final proof term are also labeled as positive examples.	P

3.1.1 Procedural Features

The first category of feature is the set of procedural features, properties of the solving process that can be quickly calculated within cvc5. Several of these features were explained in Section 2.1. In this category are the *age* and *phase* of the term. In addition to these, the features *varFrequency*, *tried* and *depth* are used. *VarFrequency* indicates how many times the variable (V) under consideration appears in the quantified expression (Q). The *tried* feature indicates how many times this term (T) was already tried within this quantified expression. The *depth* is simply the syntactic depth of the term.

3.1.2 Bag of Words (BOW) Features

The Bag of Words features represent the number of occurrences of symbols in the given quantified expression and term. In our design, uninterpreted symbols are treated anonymously and interpreted symbols non-anonymously. This means that any interpreted symbol, such as $+$, will get its own feature whose value is the number of occurrences of the symbol in the formula. Uninterpreted symbols, i.e., those that were introduced by the user are collapsed into representative classes. For instance, all function symbols are represented by one class. The numerals (interpreted constants) are a special case. While these are interpreted, there are infinitely many of them and they therefore cannot have separate features, and therefore are also collapsed into a single nodetype in this feature category. However, in the *Numeral Features* (Section 3.1.4), we deal with numerals in a more semantic way.

For the calculation of BOW we use the abstract syntax tree (AST) of cvc5. For every symbol appearing in terms and formulas cvc5 determines its *kind*. These kinds include, e.g., *variable*, *skolem*, *not*, *and*, *plus*, *forall*, and many others. We use these syntactic kinds to define a *bag-of-words*-type featurizer $\mathrm{BOW}(x)$, where x is a term or a quantified formula, and the information returned by BOW consists of counts of kinds of symbols appearing in x. For example, $\mathrm{BOW}(\forall x\,(2 + x = \mathrm{skl}_1 + 3)) = \{\mathit{forall} : 1, \mathit{variable:\ 1}, \mathit{const} : 2, \mathit{skolem} : 1, \mathit{plus} : 2\}$. Non-occurring kinds are set to 0 in this representation. We remark that cvc5 represents formulas as directed acyclic graphs, rather than trees, which is also reflected here, i.e., any repeated sub-formula is counted only once.

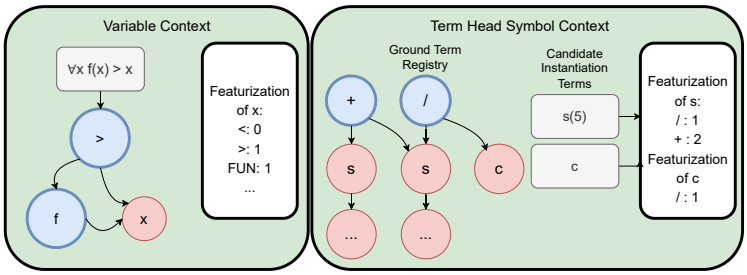

Figure 3 Schematic representation of the context features. The variable context aggregates which symbols occur as the parent of the variable in the current quantified expression. The term head symbol context aggregates which symbols occur in the ground term registry as parents of the head symbol of the candidate term under consideration.

3.1.3 Context Features

The third category of features is the set of context features (see Figure 3). There are two types of context features used. The first is the *variable context*, which aggregates information about which symbols are used as parents of the variable in the current quantified expression. The second type is the *term head symbol context*, which contains information about which symbols are parents of the head symbol of the current candidate term in the ground term registry of the solver: this gives information about the whole problem, even across quantified expressions. The feature vectors are sparse, in the sense that most nodetypes will not show up in every context: most of the possible features will be 0. In the figure we have simplified mostly to the symbols that do appear, but for the ML predictor many features are 0.

3.1.4 Numeral Features

While in the BOW Features the numerals in the problems are all mapped to one single nodetype, this is not optimal. Especially because the benchmark problems we test on are integer arithmetic problems, giving the machine learning component some information about which numbers are in the formula should be useful. To allow the machine learning system to do some elementary comparison operations on the terms it needs to decide the score of, we add 4 additional features, which are the *minimum* and *maximum* number in the current quantified expression (Q) and the current candidate term (T). When there are no nodes of numeral type in Q or T, we fill in the features with a fixed combination of numbers where the minimum is higher than the maximum, so that it can be distinguished from the rest of the cases.

3.1.5 Parent Label Propagation

The last setting in our algorithm concerns the *parents* of the proof terms. Note that this is a different kind of setting than the features before. Here we are concerned about which candidate terms are counted as positive examples (that is to say, for which the score function should predict 1) and which are negative examples (for which the prediction should be 0).

When the base solver is run and produces a proof, a set of instantiations is obtained that leads to a contradiction. However, these terms are created by iterations of the enumeration loop, creating more and more terms from the Herbrand universe. It may be the case that these final instantiations cannot be created in one iteration. We may need more than 1 instantiation to create the final instantiated term that solves the problem. For example

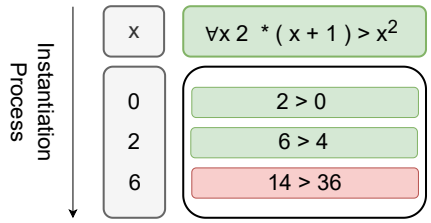

Figure 4 Example instantiation process that shows the difference between a term that constitutes a proof and the process of getting to the point where it is possible to instantiate that term within the cvc5 enumeration setup.

in Figure 4, we see that although instantiating with $x \mapsto 6$ will expose the contradiction, the ground term 6 is not available at the beginning of solving. Instantiating with 0 at the beginning makes the term 2 available. Instantiating with 2 makes the term 6 available, which leads to a contradiction. To reflect this, we propagate the positive label of these instantiations (i.e. 6) to the parent instantiations, i.e., the instantiations that were done to create the final instantiation candidate terms, 2 and 0 in the example from Figure 4.

4 Experimental Evaluation

The SMT solver we augment with ML-guided enumerative instantiation is cvc5. It implements multiple techniques for quantifier instantiation (see Section 2). Due to the inherent difficulty of the overall problem, it is not the case that one technique would be better than another one. On the contrary, the techniques exhibit a high degree of orthogonality in terms of the number of solved instances [36, 19]. Therefore, it is meaningful to focus on improving the techniques independently of one another. In our scenario, we let the solver use the enumeration technique combined with the relevant domain heuristic (Section 2.1); all the other techniques are explicitly turned off, which also lets us more clearly isolate the effect of machine learning guidance.

We refer to cvc5 with this baseline strategy as the *base solver*. We remark that this setup alone is already a very strong solver, often outperforming more complex techniques, as shown in the literature [19]. The objective of the evaluation is to compare the base solver with the base solver augmented with ML guidance.

A crucial prerequisite to training a strong ML model is to have a sizeable, high-quality set of training examples. They can be collected by running the solver on available problems and by recording which instantiations were *positive* (appeared in a proof) and which were *negative* (redundant).

Note that the notion of a *positive or negative example* is not strict here since a single problem may have multiple alternative proofs resulting in different sets of *positives* and *negatives*. Moreover, the solver may arrive at a given proof in multiple ways, performing the same set of useful instantiations but different sets of redundant ones, which results in a different collections of negative examples.

Based on these observations, it is clear that in order to collect a rich and illustrative set of training examples, it is beneficial to run the solver on a given set of problems multiple times in varied ways, which will result in multiple alternative proofs and proof searches. Thus, we establish the following methodology for collecting the training data. First, an unguided solver is run on a given set of problems. The data recorded from these proof attempts give an initial set of training examples, which is used to train an initial ML-model. Then, the

Algorithm 1 Incremental solving-training feedback-loop ended with solving holdout problems.

Require: target problems: P_{target}, holdout problems: P_{holdout}, number of iterations: N, grid of hyper-parameters: H_{grid}

1: $M \leftarrow \{\}$ ▷ empty initial machine-learning model
2: $D \leftarrow \{\}$ ▷ empty initial set of training examples
3: **for** $i \leftarrow 0$ *to* $N-1$ **do**
4: $L \leftarrow \textsc{Solve}(P_{\text{target}}, M)$ ▷ solve target problems, save proofs and statistics
5: $D \leftarrow D \cup \textsc{ExtractTrainingExamples}(L)$ ▷ update training data
6: $H \leftarrow \textsc{GridSearch}(D, H_{\text{grid}})$ ▷ find a good set of training hyper-parameters
7: $M \leftarrow \textsc{TrainModel}(D, H)$ ▷ train a new model on all training examples
8: $\textsc{Solve}(P_{\text{holdout}}, M)$ ▷ solve holdout problems

solver – this time guided by the ML-model – is run again on the problems. This gives new training examples augmenting the database. Solving and training may be interleaved an arbitrary number of times. It constitutes a positive feedback loop – in each iteration, the solver is guided by a new, different, and hopefully stronger ML-model which results in a growing and varied training set. A similar looping-style approach was already used in the context of automated theorem proving [33, 41].

In the evaluation, we focus on two separate, but equally important, goals:

1. the *cumulative goal*: solve automatically as many of the problems as possible over time. This is done by running the ML-guided solver multiple times over them and improving it by training the ML model on data collected across the runs. In this setting we gradually solve more problems, that the base solver could not prove.
2. the *single-instance goal*: evaluate the ability of the learning model to improve the solver on unseen problems.

The importance of the single-instance goal is clear—this is how traditionally improvement in SMT is measured, i.e., how many more problem instances are solved. Here we emphasize that the cumulative goal is just as important. Indeed, in many cases users wish to solve a group of formulas and are happy to leave the solver work on them for an extended period of time. This is particularly true for groups of *similar* formulas. This might be the case for example for verification conditions coming from a certain piece of software that is being verified. In such scenarios, the user is not interested if the SMT solver solves many instances in a competition but is interested in how many instances are solved from this particular set.

Note that in a traditional setting it is unclear how to improve on the cumulative goal. In contrast, an ML-guided SMT solver naturally has the opportunity to generalize from previously solved (easier) problems to the harder ones.

4.1 Experimental Setting

To assess the performance of our method for both these goals, we use the looping-style approach described above, additionally splitting the initial set of problems into *target set* and *holdout set*. The problems from the targets that are used to gradually collect training examples for training the ML model to be used in the next iteration. The cumulative goal is measured by how many more instances are gradually sold from the target set. The single-instance goal is measured by the base solver with the solver guided by the ML-model all obtained in the last iteration.

The looping procedure is presented in Algorithm 1. The function Solve(P, M) runs the solver over problems in P, using the ML-model M for guidance. When $M = \{\}$, in the initial round 0, Solve(P, M) runs the solver with the standard, age-based ordering of the terms in place of the ML-guidance. In the experiments, the number of iterations N is set to 20.

The used ML model (LightGBM) has multiple hyper-parameters governing its training, which potentially significantly influence the predictive performance of the model and therefore should be tuned [43]. We fix a set of several important parameters and candidate values for them (H_{grid}). These are the following: `learning_rate`: 0.01, 0.05, 0.1, `num_leaves`: 16, 64, 256, `max_bin`: 16, 64, 256.

After each update of the training set in the loop, a grid-search is used (function GridSearch) to establish the best hyper-parameters (H) for the next training (according to the AUC metric [14] on a random subset of validation examples). The number of trees in LightGBM model is an important parameter, however we do not include this parameter in the grid search and just fix its value to 100. Increasing the number of trees typically improves the "offline" ML performance metrics, however, it also slows down producing the predictions, which in turn may decrease the number of solutions found within a given time limit.

A large majority of the instantiations tried during the proof attempts are redundant, which results in a significant disproportion between numbers of the positive and the negative examples being collected. To expose the ML-model more to the positives, we under-sample the negatives so that its number is kept below 10 $\times$ number of positives.

In experiments, the ML-guided solver is compared to the *base solver*. However, when considering the cumulative number of problems solved across multiple iterations, one should investigate whether the extra problems solved are really due to the learned strategy and not only due to the randomness injected into the process. Thus, to perform an ablation study, we additionally compare the ML-guided solver with a *randomized solver*. It is the same as the base solver with the following exception: it uses the predefined, age-based ordering additionally swapping each term randomly with a term next to it in the ranking with the probability 0.1. This parameter is selected heuristically: we want to have a solver which is similar to the well-performing, base solver, and at the same time is non-deterministic to some degree. Our initial experiments also indicate that deviating too much from the age-based ordering is detrimental to the solver: choosing a totally random order leads to a significant decrease in the number of solved instances (around 30%).

In experiments, we fix a timeout of 60 s per one proof attempt for all the solvers. Note that the ML-guided solver spends a non-negligible amount of time just on producing predictions from the ML-model, which means that it will be able to perform fewer steps in its proof searches than the unguided solver. However, to have a realistic evaluation scenario, we give the same timeout for each solver, with the outlook that the ML-guided solver will compensate for the slowdown with its learned strategy.

4.2 Data for Evaluation

For evaluation, we use six benchmarks from SMT-LIB [5]: (1) UFLIA boogie, (2) UFLIA grasshopper, (3) UFLIA tokeneer, (4) UFNIA sledgehammer, (5) UFNIA Preiner, (6) UFNIA vcc havoc.

UFNIA and UFLIA refer here to two different SMT logics: non-linear and linear integer arithmetic, respectively, with uninterpreted function symbols. (1) originates from various problems from formal verification formulated in an intermediate verification language Boogie [3]. (2) is a benchmark derived from a software verification project concerning heap-manipulating programs [35]. (3) was derived from a security verification project for

biometric identification software [24]. (4) originates from Sledgehammer, a component of the Isabelle/HOL interactive theorem prover that enables applying SMT to discharge goals arising in interactive proofs [8]. The Sledgehammer problems come from various areas of mathematics and computer science. (5) was a project on verifying rewriting rules for bit-vectors irrespective of bit-width [28]. (6) are benchmarks taken from the VCC C program verifier [10] and HAVOC [42], a heap-aware verifier for C programs.

Some of the problems from these benchmarks may be solved without performing any instantiations. They are filtered out as not relevant for our evaluation. Then, the sizes of the benchmarks are: UFLIA boogie: 1005, UFLIA grasshopper: 382, UFLIA tokeneer: 257, UFNIA sledgehammer: 1329, UFNIA Preiner: 3897, UFNIA vcc havoc: 760. Each of the benchmarks is randomly split into target and holdout parts (P_{target}, P_{holdout}) of sizes 75% and 25%, respectively.

4.3 Results and Discussion

This section presents the results of the evaluation of the ML-guided solver with one initial solving iteration performed by the unguided, base solver, and 19 training-solving iterations. Because of the non-deterministic nature of the training procedure, each loop with the ML-guided and the randomized solvers is run 3 times and the presented results are averaged.

The presentation of the results is divided into three subsections. The first two are concerned with the cumulative and single-instance goals (see introduction to Section 4). The last subsection presents an ablation study evaluating the importance of the different groups of features (see Subsection 3.1).

4.3.1 Cumulative Goal

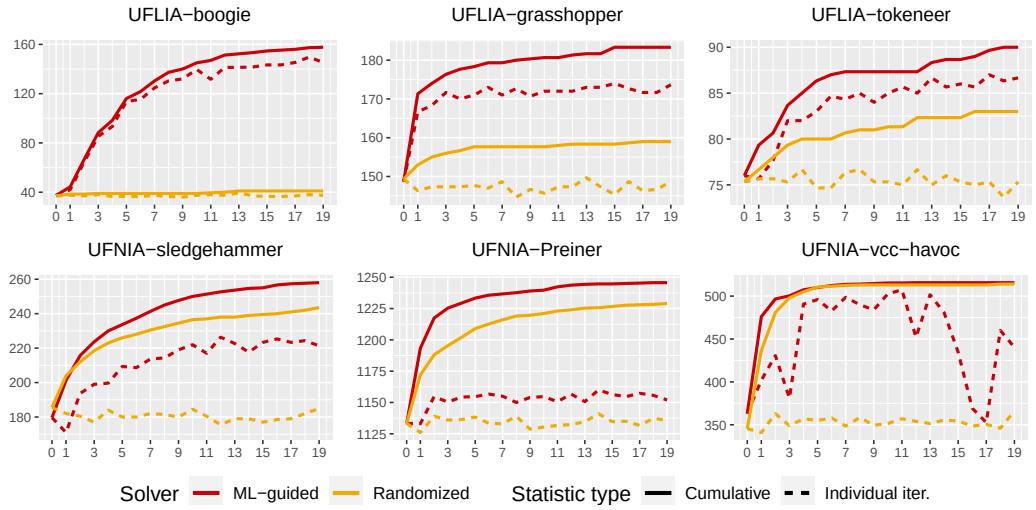

Figure 5 Numbers of problems solved (y-axis) in the looping evaluation across twenty iterations (x-axis) for six benchmarks. Dashed lines refer to numbers of problems solved in a given iteration; solid lines refer to cumulative number of problems solved in a given iteration and all past iterations.

Figure 5 shows the number of solved instances for the considered families across the iterations of the loop (see Algorithm 1). All, except for the last two, families demonstrate a clear advantage of using ML-guidance. When focusing on the cumulative goal (solid lines),

Table 2 Target problems solved cumulatively across all 20 iterations of the training-solving loop by the randomized and the ML-guided solvers.

	Randomized	ML-guided	Improvement (%)	Number of problems
UFLIA boogie	41.0	157.6	284.4	754
UFLIA grasshopper	159.0	183.3	15.3	287
UFLIA tokeneer	83.0	90.0	8.4	193
UFNIA sledgehammer	243.5	258.0	6.0	997
UFNIA Preiner	1228.3	1245.7	1.4	4776
UFNIA vcc havoc	513.0	515.7	0.5	570

both the ML-guided and the randomized solver exhibit diminishing returns—eventually they plateau. However, in the case of a randomized solver, a plateau occurs typically far earlier than in the ML-guided case. This is likely explained by the ability of the ML-guidance to keep inventing new approaches inspired by newly solved problems. In contrast, randomization very quickly hits the wall since the original heuristic used by the base solver is already good. This is further supported by the observation that the learned ML-model solves an increasing number of the overall instances, whereas the randomized one solves roughly the same number of problems in every iteration. More detailed numbers can be found in Table 2.

The last two families (UFNIA-Preiner, UFNIA-vcc-havoc) do not show any significant improvements with ML-guidance. Possibly, this might be that we are simply too close to the limits of what the solver can do in this configuration and other quantifier instantiation methods need to be also considered (see discussion on the future work in Section 6). On the contrary, UFLIA-boogie shows an exceptional improvement with ML-guidance and shows no improvement by randomization.

4.3.2 Single-Instantiation Goal

Here we compare the base solver with the ML-guided solver using the ML-model obtained in the last iteration of the evaluation loop. These results are calculated on the *holdout set*—meaning, on a set of problems that were **not** used for training of the ML-model. Two types of metrics are considered. First, we consider the number of instantiations that the solver needed to do to solve the given problem—effectively, this is the *abstract time*, measuring the quality of the guidance. Second, we consider the actual CPU time needed to solve the problem. Figure 6 shows the results for these two metrics in two separate scatterplots. More detailed numbers can be found in Table 3.

Table 3 Holdout problems solved by the base and ML-guided solvers.

	Base	ML-guided	Improvement (%)	Number of problems
UFLIA boogie	11	43.0	290.9	251
UFLIA grasshopper	59	66.6	12.9	95
UFLIA tokeneer	25	30.0	20.0	64
UFNIA sledgehammer	63	59.3	-5.8	332
UFNIA Preiner	383	394.3	3.0	1592
UFNIA vcc havoc	175	145.6	-16.8	190

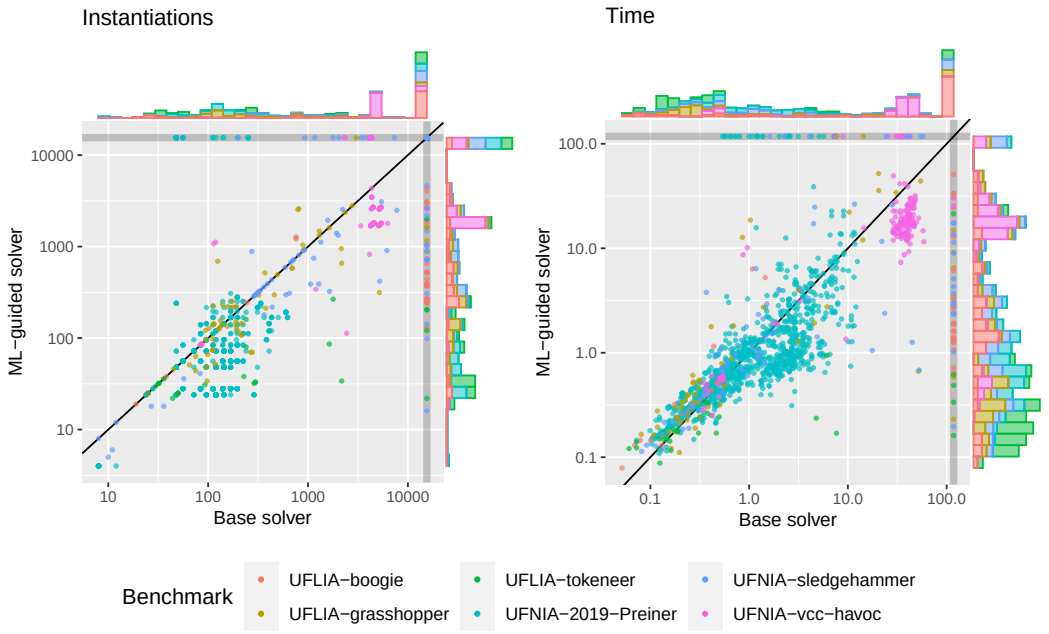

Figure 6 Comparison of number instantiations (left) and solving times (right) in log scale, where each point is a testing problem. Points in dark gray stripes were solved by only one of the solvers.

The results for the number of instantiations clearly speak for ML-guidance as vast majority of the points are below the diagonal. Further, the histograms for the ML-guided solver shows a more even distribution of values, whereas the base solver is mainly stacked on timeouts. In the case of time-evaluation, we still see a large portion of the instances under the diagonal and the histogram is also more tilted towards lower values. However, we also do see some worsening in terms of time. This is not entirely surprising because ML-guidance comes at a price because the ML-model needs to be evaluated for each considered term in every instantiation step. This is also a likely explanation for the worsening in the havoc family. Nevertheless, given the highly positive results in terms of number of instantiations, better engineering of the ML guidance has the potential of further improving these results.

4.3.3 Ablation Study

In this subsection we look at the importance of the different types of features that were used to train the ML-model (see Subsection 3.1). For this we consider the boogie family where ML-guidance had the most effect and therefore enables us to clearly observe the effect of the different features. Since it would be impractical to try all combinations of the features, we use the standard ablation approach, i.e., removing one feature type at a time and observe the effect of this removal. The results are shown in Figure 7 and Table 4. The ablation study relates two very clear messages. Firstly, the full set of features outperforms all the other configurations. Importantly, the full set of features is also the best one on the unseen problems (holdout set). Secondly, the performance is significantly worse without the procedural features. This is definitely an interesting observation because the procedural features depend on the previous decisions of the solver, e.g., the number of times a term has been used so far. This observation indicates that is important for the ML-guidance to "understand" its old decisions and therefore serves as a guideline for future research.

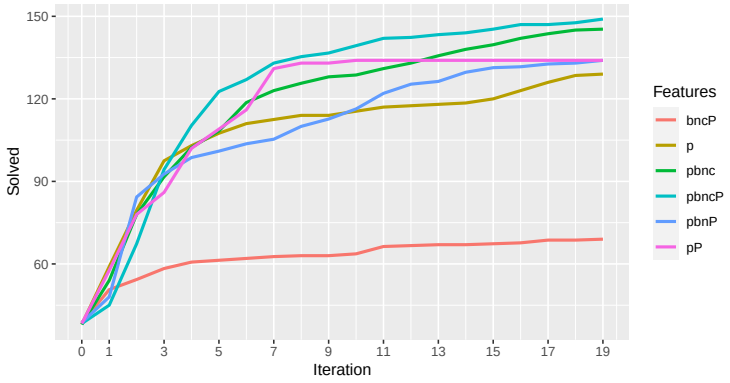

Figure 7 Ablation analysis for different sets of features. The dataset used here is UFLIA boogie, and the solid lines refer to the cumulative numbers of solved problems. The letters in the names of runs refer to abbreviations in Table 1.

Table 4 UFLIA boogie problems solved with the ML-guided solver using different sets of features.

Features	Solved cumulatively, in target	Solved, in holdout
bncP	69.0	13.3
p	126.6	21.0
pP	132.6	27.6
pbnP	134.0	32.0
pbnc	145.3	33.3
pbncP	149.0	36.0

4.3.4 Training and Predicting Time

The time of training a single LightGBM model is negligible in comparison to the solving time and it is in the order of minutes (it grows as more training examples are collected, but it was below 10 minutes for all benchmark sets and iterations).

The total time required to run a full loop of 20 training-solving iterations (including the hyper-parameter tuning) on 1 family depends on the number and complexity of problems in each set. We ran all the loops parallelizing across 20 cores, and the total time for running one loop was between 3 and 30 hours. To optimize this, one could apply early stopping when observing diminishing returns.

5 Related Work

In recent years, machine learning has been widely applied in automated theorem proving. Both gradient boosted trees and graph neural networks have been applied for premise selection and guidance of the automated theorem prover E [18, 34] as well as in a reinforcement learning setting for connection-style provers [20]. ML guidance was also used in the context of SAT solving [40, 25]. In the context of SMT, ML has been mostly used *outside* of the solver. ML advice was used to predict the best SMT solver out of a given portfolio and problem [39, 32]. Similarly, FastSMT uses ML to design strategies for the SMT solver Z3 [1], where the BOW representation shows to be most successful, strengthening our choice of this representation.

An unpublished technical report by Ouraou et al. [31] describes an attempt to apply ML for quantifier instantiation in the SMT solver VeriT [9]. The report concludes that the attempt was overall unsuccessful. This approach filters out instantiation terms deemed redundant by the ML model. Further, a different set of features, more expensive, is considered. In contrast, we apply stable ordering on the existing terms, which enables us to piggyback on the existing good performance of the solver. Indeed, in our approach, if ML scores some term candidates equally, they are kept in the same order as in the base solver and candidates are never removed from the pool. Our choice of features lets us calculate quickly the ML predictions online without being detrimental to the solving time.

6 Conclusions and Future Work

The paper designs an ML guidance of quantifier instantiation in the context of SMT for problems with quantifiers. Quantifiers are a particularly interesting target for ML because they typically cause undecidability and therefore represent an inherent challenge for automated solvers. In the presented approach, the ML advice influences the solver by ordering the candidate terms to be considered for quantifier instantiations. The right choice of instantiations is crucial for solving with quantifiers. Indeed, one particular formula is often solved by a handful of instantiations in one ordering and it times-out after hundreds of thousands of instantiations in another. The challenge we are facing here is both conceptual and technological. At the conceptual level, the right set of features needs to be designed. At the technological level, we need an integration of ML predication into the solver that does not hinder the performance of the solver (ML prediction is run on each candidate term).

The experimental evaluation shows that our approach rises to the challenge. When run on a set of formulas, cumulatively ML-guidance enables solving significantly more problems than randomizing the solver. Improvements are also seen on a holdouts set (a set on which the solver was not trained). ML advice enables us to solve more problems and reduce the number of instantiations needed. The effect is most pronounced in the considered *boogie* benchmark, where the final ML-model enables to solve nearly 3 times more testing problems, and during training it accumulates more than 3 times more solved instances compared to a randomized solver. We also achieve improved performance on the *grasshopper, sledgehammer and tokeneer* benchmarks. In some families, we have seen worsening in the holdout set, which could partially be explained by the CPU time overhead of running ML prediction. This indicates that it would pay off to better engineer the predictor so that this overhead is reduced. In an ablation study on the *boogie* benchmark, we show that each of our feature categories contributes to the final results.

This paper shows that ML has the potential of boosting SMT solving and it opens a number of opportunities for future work. Further ML models may be proposed for specific logics (our method is generic). Direct interaction between quantifiers could be taken into account. Rather than predicting an order of terms on a single variable, the ML model could predict good combinations for tuples of variables. Last but not least, ML advice could be applied to the other quantifier instantiation methods that are in use in the SMT field.

References

1 Mislav Balunovic, Pavol Bielik, and Martin T. Vechev. Learning to solve SMT formulas. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8,*

2018, Montréal, Canada, pages 10338–10349, 2018. URL: https://proceedings.neurips.cc/paper/2018/hash/68331ff0427b551b68e911eebe35233b-Abstract.html.

2 Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS*. Springer, 2022. doi:10.1007/978-3-030-99524-9_24.

3 Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO*, volume 4111, pages 364–387. Springer, 2005. doi:10.1007/11804192_17.

4 Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification – 23rd International Conference, CAV*, volume 6806, pages 171–177. Springer, 2011. doi:10.1007/978-3-642-22110-1_14.

5 Clark W. Barrett, Leonardo Mendonça de Moura, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The SMT-LIB initiative and the rise of SMT - (HVC 2010 award talk). In *Hardware and Software: Verification and Testing - 6th International Haifa Verification Conference, HVC*, volume 6504, page 3. Springer, 2010. doi:10.1007/978-3-642-19583-9_2.

6 Clark W. Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018. doi:10.1007/978-3-319-10575-8_11.

7 Nikolaj Bjørner and Mikoláš Janota. Playing with quantified satisfaction. In *20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning – Short Presentations, LPAR*, volume 35, pages 15–27. EasyChair, 2015. URL: https://easychair.org/publications/paper/jmM.

8 Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. *J. Autom. Reason.*, 51(1):109–128, 2013. doi:10.1007/s10817-013-9278-5.

9 Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. veriT: An open, trustable and efficient SMT-solver. In *Automated Deduction – CADE-22, 22nd International Conference on Automated*, volume 5663, pages 151–156. Springer, 2009. doi:10.1007/978-3-642-02959-2_12.

10 Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. Vcc: Contract-based modular verification of concurrent c. In *2009 31st International Conference on Software Engineering-Companion Volume*, pages 429–430. IEEE, 2009.

11 David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005. doi:10.1145/1066100.1066102.

12 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*, 2019. doi:10.18653/v1/n19-1423.

13 Azadeh Farzan and Zachary Kincaid. Strategy synthesis for linear arithmetic games. *Proc. ACM Program. Lang.*, 2(POPL):61:1–61:30, 2018. doi:10.1145/3158149.

14 Tom Fawcett. An introduction to ROC analysis. *Pattern Recognit. Lett.*, 27(8):861–874, 2006. doi:10.1016/j.patrec.2005.10.010.

15 Michael J. Fischer and Michael O. Rabin. Super-exponential complexity of presburger arithmetic. In *Texts and Monographs in Symbolic Computation*, pages 122–135. Springer Vienna, 1998. doi:10.1007/978-3-7091-9459-1_5.

16 Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiability modulo theories. In *Computer Aided Verification, 21st International Conference, CAV*, pages 306–320, 2009. doi:10.1007/978-3-642-02658-4_25.

17 Jacques Herbrand. *Recherches sur la théorie de la démonstration*. Doctorat d'état, La Faculté des Sciences de Paris, 1930.

18 Jan Jakubův, Karel Chvalovský, Miroslav Olšák, Bartosz Piotrowski, Martin Suda, and Josef Urban. ENIGMA Anonymous: Symbol-independent inference guiding machine (system description). In *Automated Reasoning – 10th International Joint Conference, IJCAR*, volume 12167, pages 448–463. Springer, 2020. doi:10.1007/978-3-030-51054-1_29.

19 Mikoláš Janota, Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. Fair and adventurous enumeration of quantifier instantiations. In *Formal Methods in Computer-Aided Design*, 2021.

20 Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. Reinforcement learning of theorem proving. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems, NeurIPS*, 2018. URL: https://proceedings.neurips.cc/paper/2018/hash/55acf8539596d25624059980986aaa78-Abstract.html.

21 Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*, pages 3146–3154, 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/6449f44a102fde848669bdd9eb6b76fa-Abstract.html.

22 Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems*, pages 1106–1114, 2012. URL: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

23 James Donald Monk. *Mathematical logic*, volume 37. Springer Science & Business Media, 2012.

24 Yannick Moy and Angela Wallenburg. Tokeneer: Beyond formal program verification. *Embedded Real Time Software and Systems*, 24, 2010.

25 Saeed Nejati, Ludovic Le Frioux, and Vijay Ganesh. A machine learning based splitting heuristic for divide-and-conquer solvers. In *Principles and Practice of Constraint Programming – 26th International Conference, CP*, volume 12333, pages 899–916. Springer, 2020. doi:10.1007/978-3-030-58475-7_52.

26 Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark W. Barrett, and Cesare Tinelli. Solving quantified bit-vectors using invertibility conditions. In *Computer Aided Verification – 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018*, volume 10982, pages 236–255. Springer, 2018. doi:10.1007/978-3-319-96142-2_16.

27 Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark W. Barrett, and Cesare Tinelli. Syntax-guided quantifier instantiation. In *Tools and Algorithms for the Construction and Analysis of Systems – 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS*, volume 12652, pages 145–163. Springer, 2021. doi:10.1007/978-3-030-72013-1_8.

28 Aina Niemetz, Mathias Preiner, Andrew Reynolds, Yoni Zohar, Clark Barrett, and Cesare Tinelli. Towards bit-width-independent proofs in SMT solvers. In *International Conference on Automated Deduction*, pages 366–384. Springer, 2019.

29 Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006. doi:10.1145/1217856.1217859.

30 Eniola Olaleye. How to win any ML contest, 2021. Published as https://medium.com/machine-learning-insights/how-to-win-any-ml-contest-244a12c62f30.

31 Danieli El Ouraou, Pascal Fontaine, and Cezary Kaliszyk. Machine learning for instance selection in SMT solving, 2019. URL: https://members.loria.fr/delouraoui/links/instanceselection_paper.pdf.

32 Nikhil Pimpalkhare, Federico Mora, Elizabeth Polgreen, and Sanjit A. Seshia. Medleysolver: Online SMT algorithm selection, 2021. doi:10.1007/978-3-030-80223-3_31.

33 Bartosz Piotrowski and Josef Urban. ATPboost: learning premise selection in binary setting with ATP feedback. In *Automated Reasoning – 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC*, volume 10900, pages 566–574. Springer, 2018. doi:10.1007/978-3-319-94205-6_37.

34 Bartosz Piotrowski and Josef Urban. Stateful premise selection by recurrent neural networks. In *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, volume 73, pages 409–422. EasyChair, 2020. URL: https://easychair.org/publications/paper/g38n.

35 Ruzica Piskac, Thomas Wies, and Damien Zufferey. GRASShopper. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014.

36 Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. Revisiting enumerative instantiation. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 10806, pages 112–131, 2018. doi:10.1007/978-3-319-89963-3_7.

37 Andrew Reynolds, Tim King, and Viktor Kuncak. Solving quantified linear arithmetic by counterexample-guided instantiation. *Formal Methods Syst. Des.*, 51(3):500–532, 2017. doi:10.1007/s10703-017-0290-y.

38 Andrew Reynolds, Cesare Tinelli, and Leonardo Mendonça de Moura. Finding conflicting instances of quantified formulas in SMT. In *Formal Methods in Computer-Aided Design, FMCAD*, pages 195–202. IEEE, 2014. doi:10.1109/FMCAD.2014.6987613.

39 Joseph Scott, Aina Niemetz, Mathias Preiner, Saeed Nejati, and Vijay Ganesh. MachSMT: A machine learning-based algorithm selector for SMT solvers. In *Tools and Algorithms for the Construction and Analysis of Systems – 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS*, volume 12652, pages 303–325. Springer, 2021. doi:10.1007/978-3-030-72013-1_16.

40 Daniel Selsam and Nikolaj Bjørner. Guiding high-performance SAT solvers with unsat-core predictions. In Mikoláš Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing – SAT 2019 – 22nd International Conference, SAT 2019, Lisbon, Portugal*, volume 11628, pages 336–353. Springer, 2019. doi:10.1007/978-3-030-24258-9_24.

41 Josef Urban, Geoff Sutcliffe, Petr Pudlák, and Jiří Vyskočil. MaLARea SG1 – machine learner for automated reasoning with semantic guidance. In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008*, volume 5195, pages 441–456. Springer, 2008. doi:10.1007/978-3-540-71070-7_37.

42 Julien Vanegue and Shuvendu Lahiri. Towards practical reactive security audit using extended static checkers. In *IEEE Symposium on Security and Privacy (Oakland'13)*, May 2013. URL: https://www.microsoft.com/en-us/research/publication/towards-practical-reactive-security-audit-using-extended-static-checkers/.

43 Soner Yildirim. How to tune the hyperparameters for better performance, 2020. Published as https://towardsdatascience.com/how-to-tune-the-hyperparameters-for-better-performance-cfe223d398b3.