MachSMT: A Machine Learning-based Algorithm Selector for SMT Solvers

Joseph Scott[1], Aina Niemetz[2], Mathias Preiner[2], and Vijay Ganesh[1]

[1] University of Waterloo, Ontario, Canada
{joseph.scott,vijay.ganesh}@uwaterloo.ca
[2] University of Stanford, Stanford, USA
{niemetz,preiner}@cs.stanford.edu

Abstract. In this paper, we present MachSMT, an algorithm selection tool for state-of-the-art Satisfiability Modulo Theories (SMT) solvers. MachSMT supports the entirety of the logics within the SMT-LIB initiative. MachSMT uses machine learning to learn empirical hardness models (a mapping from SMT-LIB instances to solvers) for state-of-the-art SMT solvers to compute a ranking of which solver is most likely to solve a particular instance the fastest. We analyzed the performance of MachSMT on 102 logics/tracks of SMT-COMP 2019 and observe that it improves on competition winners in 49 logics (with up to 240% in performance for certain logics). MachSMT is clearly not a replacement for any particular SMT solver, but rather a tool that enables users to leverage the collective strength of the diverse set of algorithms implemented as part of these sophisticated solvers. Our MachSMT artifact is available at https://github.com/j29scott/MachSMT.

Keywords: SMT Solvers · Machine Learning · Algorithm Selection

1 Introduction

Satisfiability Modulo Theories (SMT) solvers are tools that decide the satisfiability of formulas over first-order theories, such as bit-vectors, floating-point, integers, reals, strings, arrays and their combinations [14, 5, 19, 13, 34, 16]. SMT solvers have had revolutionary impact on applications in software engineering (broadly construed), such as software testing [12, 35], verification [18, 11, 29, 30], and AI [26]. There is an insatiable demand for more efficient and expressive solvers, especially as researchers find new application domains (e.g., verification and synthesis of cryptographic primitives [8]) and attempt to scale existing applications to larger settings (e.g., automatic bug finding in large commercial software [21, 3]).

In response to this high demand, the SMT community has developed a plethora of solvers. For example, in the 2019 edition of the annual SMT-COMP competition [6, 22], more than 50 solvers and configurations of which were submitted. Many of these implement very different algorithms to tackle the satisfiability problem for (a combination of) theories, with significantly varying performance profiles. For example, for the quantifier-free theory of floating-point alone there exist several substantially different decision procedures, e.g., bit-blasting [10], abstract CDCL [9], inter-reduction methods [40], and reduction to global optimization [17, 27]. In this particular example, input instances can be derived from a variety of applications, such as software verification or analysis of machine learning (ML) models [41]. In such a scenario, a very natural question arises: which solver and algorithm is best to use for a given input instance?

Another well known issue with many SMT solvers (even state-of-the-art ones) is that users may not know a priori which formula features would make an instance easy or hard to solve. This can be very frustrating for users as they have to try combinatorially many different encoding and solver pairs before they can figure out which combination works best for their specific scenario. Users have also noted that as their application needs change, what was once a great solver in an older setting is suddenly not very good in the newer one.

One way to address the above-mentioned problems is to use an automated algorithm-selection tool that can automatically and with high accuracy select the optimal SMT solver from a set of solvers, for a given SMT formula. A side-effect of such a tool is that it may be helpful for solver developers to identify theory features or encodings that are the root cause of inefficiency in their solvers.

To this end, we introduce MachSMT, a machine learning-based algorithm-selection tool. MachSMT supports the entirety of the theories and logics within the SMT-LIB initiative [7]. MachSMT constructs machine learned models by analyzing the runtimes of solver configurations on benchmarks with respect to the frequencies of grammatical constructs (i.e., of predicates, functions, rounding modes, etc.) of its inputs. MachSMT then takes as input an instance for a specified theory of interest and outputs the solver or a ranking of solvers as predicted to have the lowest runtime.

Brief Overview of the Design of MachSMT: At a high-level, MachSMT works as follows: For a particular combination of solver, logic and and its runtime data on benchmarks in this logic, our tool MachSMT uses machine learning to construct an empirical hardness model (a mapping from instances to solver runtimes), specifically via the use of an adaptive boosting regressor [15] with principal component analysis (PCA) [47, 23]. From the user's perspective, for any given input, MachSMT constructs a list of predicted log runtime for each solver. MachSMT then returns this ranking or the fastest solver according to this ranking. MachSMT is clearly not a replacement for any particular SMT solver, but rather a tool that enables users to leverage the collective strength of the diverse set of algorithms implemented as part of these sophisticated solvers.

While algorithm selection has been considered in the broad setting of solvers (e.g., QBF solvers [38] and SAT solvers [48]) as well as certain specific SMT theories and applications [42, 4, 46], we are not aware of previous work on algorithm selection aimed at the entirety of SMT-LIB. Our results show that the MachSMT algorithm selector is effective and highly accurate, in that it outperforms the competition winners on several tracks from the SMT-COMP 2019 competition. Additionally, MachSMT is easy to use and extend by users.

Contributions

We make the following contributions in this paper:

1. **The MachSMT Tool**: We present the MachSMT tool, an algorithm selection tool for the entirety of SMT-LIB. MachSMT learns a machine learning (ML) model for algorithm selection for each logic and track within the SMT-LIB initiative and SMT-COMP. Our MachSMT tool is designed to be easily tuned and extended by SMT solver users for any logic and/or applications of their interest. (Section 3)

2. **Experimental Analysis of MachSMT**: We perform a full experimental analysis of the MachSMT algorithm selection tool. We analyzed the performance of MachSMT in all logics of all tracks in SMT-COMP, and observe that it improves on competition winners in 49 (out of a total of 102) divisions, with up to 240% improvement in performance for the QF_UFBV SQ and 170% for the LRA SQ division. We provide our learned models, used in our experimentation, for ease of use and transparency. While building learned models for MachSMT can be computationally expensive, installing, downloading, and using our models can be done relatively easily. (Section 4)

2 Background

A Brief Overview of Algorithm Selection: The idea of algorithm selection was first proposed and formalized by Rice [39]. Researchers have long known that given a set S of different algorithm implementations for the same specification or problem, it is often the case that one of these implementations may perform poorly on a given class of inputs while another might perform very well. This is especially true for problems believed to be computationally hard. The reasons for this phenomenon could be as diverse as choice of data structures, fundamental differences between algorithms, or the fact that heuristics implemented as part of one algorithm can exploit the input problem structure or the underlying hardware better than the other.

Users would naturally want to exploit this diversity in algorithmic approaches to difficult problems. One way to accomplish this is via the use of Empirical Hardness Models (EHM), which, given a set of features of an input instance, predict which one of the distinct algorithmic implementations is most efficient for that input. With advancements in machine learning, efficient EHMs have become a reality in many domains, especially in the broad context of logic solvers.

SMT Theories and Logics: The SMT-LIB initiative provides a standardized syntax and semantics for several first-order theories and logics [7]. In this paper, we may use the following acronyms (given in brackets), optionally in combination, to refer to various SMT-LIB logics: Quantifier Free (QF), Theory of Arrays (A), Uninterpreted Functions (UF), Bit-Vectors (BV), Floating-Points (FP), Strings (S), Integers (I), Reals (R), or mixed (IR). Further, arithmetic over I and R can be linear (LIA, LRA), or nonlinear (NIA, NRA), or difference logics

(IDL, RDL). For combinations of such, their order will appear in the order they are written above. Each theory defines various operators/functions, predicates, terms, sorts, and generalized keywords (e.g., assert, check-sat, Int, Float32). For a full list of these, as well as more details on syntax and semantics, we refer to the SMT-LIB standard [7]. We define *grammatical-constructs* or tokens as constants, functions, predicates, rounding modes (in the context of FP), sorts, and other keywords in the SMT-LIB standard.

Adaptive Boosting and Principal Component Analysis: Adaptive boosting (AdaBoost) is a *ensemble* approach to machine learning. In ensemble learning, a set of learning algorithms (e.g., weak learners) are trained, and predictions are made diplomatically across the set. In our application of AdaBoost, we use an ensemble of decision trees. For more, we refer to Drucker et al. [15]. Principal Component Analysis (PCA) is an unsupervised learning dimensionality reduction technique. PCA works by constructing a linear orthogonal transformation over a dataset. For more, we refer to Halko et al. [23, 47]. As an implementation detail, MachSMT uses scikit-learn as its ML backend [36].

3 A Description of MachSMT

Given a database of solvers, benchmarks, and their runtime data on these benchmarks, MachSMT uses machine learning to construct an empirical hardness model (EHM) [31] for each considered solver and benchmark set. In our experimental evaluation, we use the wall clock timing analysis provided by the SMT-COMP 2019 [22], which was divided into several tracks: the Single Query (SQ), Incremental (INC), Unsat Core (UC), Model Validation (MV), Challenge (CH-SQ), and Challenge Incremental (CH-INC) tracks. Each track is divided into logics (e.g., BV, LIA), and for each logic L and solver S in a track T, an EHM is learned to predict the runtime of S on all instances of L in T. Put differently, an EHM is a function that takes as input a tuple $\langle S, L, T \rangle$ and returns the natural log of the runtime $\mathcal{R}$ of solver S. We used the SMT-COMP data because it is comprehensive and available in a convenient format. However, MachSMT can be trained with any such appropriately formatted data and instances.

Note that in the INC and CH-INC tracks of SMT-COMP, every instance consists of multiple queries, and in the competition, the first deciding factor for ranking solvers in incremental tracks is the number of successfully solved queries within the time limit. As a consequence, the runtime analysis alone is not sufficient to determine a score to rank solvers, since for hard problems, solvers typically reach the time limit before all queries are solved. We thus only consider instances, where all queries were solved within the given time limit. Further, the UC and MV tracks in SMT-COMP have very specific and different objectives than the SQ and INC tracks – here the goal is not only to solve instances correctly and as fast as possible, but to provide a minimal UNSAT core (the smaller, the higher the score) in the UC track, and a correct model in the MV track. Currently, we only consider runtime to rank solvers in these two tracks. We leave more precise scores for the INC, CH-INC, UC and MV tracks to future work.

The feature set for each SMT-LIB input file is created as follows: the file is linearly scanned and the frequency of occurrence of each grammatical construct (refer the SMT-LIB standard as defined in Section 2) that appears in that file is recorded. The feature set is a set of tuples whose first element is the token name of the grammatical construct and second element is its frequency of occurrence in the input file. We use a strict 1-second time limit for all computations of features for a particular SMT-LIB input, both for building and analyzing MachSMT. If a timeout occurs, the current values state of the computed frequencies are used as features. This frequency computation is efficient and timed out only on 1.6% of the total number of SMT-LIB inputs.

each tuple of track and solver, MachSMT constructs a separate EHM, during the training phase. For certain tracks with very few benchmarks, we may add instances to that track by leveraging instances from similar logics in a different track, thus augmenting their size and improving the learning of the corresponding EHM. We explain this process using the following example: consider training an EHM for the Z3 solver for the QF_BV logic in the CH-SQ track, which has only 7 instances. We augment this training set by leveraging data from the QF_BV instances from the very similar SQ track. This gives us a much larger set of benchmarks and data points.

Further, for evaluation purposes, we deploy a k-fold cross-validation procedure (with k set to 5) to evaluate the performance of MachSMT. That is, we randomly partition the entire dataset into k bins, train on $k-1$ bins, and test on the remaining one. This process is repeated until the tool has been tested on each one of the k bins. Cross validation is commonly used as method to check whether a machine learning model can generalize over unseen data and is standard in machine learning research [37].

Before training the EHM, we apply two preprocessing steps. First, a regularizer is trained to adjust each individual feature's distribution to be zero mean and unit deviation to normalize the data. This kind of normalization is common practice in ML research and is often necessary for high classification accuracy of many ML algorithms. Second, due to the high dimensionality (148) of our feature space, we apply a well known unsupervised dimensionality reduction algorithm on the regularized data, specifically, Principal Component Analysis (PCA) [47, 23]. We use PCA to reduce the dimensionality of the feature space, which is important for the performance of the underlying ML algorithm we use. We achieved the best results with 30 dimensions.

Traditionally, algorithm selection tools such as SatZilla [49, 50] deploy linear ridge regression [24] as their primary regressor for an EHM. For MachSMT, we use Adaptive Boosting (AdaBoost) [15] based on empirical testing. Upon completion of the preprocessing, an EHM is then trained for each solver S that is competing in the track and logic of interest. When making a prediction for an input I, each solver's EHM is queried to predict a log runtime for I. We use these predictions to rank the solvers that are most likely to solve I the most efficiently.

MachSMT returns the argmin of the predicted log of runtimes. Across all logics and tracks, MachSMT maintains 765 EHMs.

We use the above process and all mentioned hyper-parameters in our training and evaluation of MachSMT. However, there are likely better configurations for a user with a particular application. To this end, we allow the user to disable individual preprocessing components, or customize the dimensionality seen by the regressor. Further, the regressor can be configured to use a variety of regression algorithms, such as Linear Regression, k-Nearest Neighbour Regression, Deep Neural-Networks, and Support Vector Machines, with ease [36].

3.1 Using MachSMT

In this subsection, we provide a brief description of how to use MachSMT. The MachSMT system includes two easy to use executable scripts: *machsmt_select* and *machsmt_build*, which run with a small set of dependencies: Python 3.5+, pip3, and a small set of pip3 installable dependencies. The executable *machsmt_select* takes as input an SMT-LIB input I and the corresponding learned EHMs and prints the name of the solver that is expected to have the lowest runtime. Our models are publicly available and can be downloaded with ease.

The second executable *machsmt_build* provides an interface to build the required learned models for *machsmt_select*. The script *machsmt_build* builds all the required EHM for the solvers, logics, and tracks of interest. It is required that the appropriate SMT-LIB benchmark dataset is available at the root of the MachSMT repository. Upon termination, the learned model is stored locally at the root of the MachSMT repository. For the entirety of SMT-LIB, we have observed this process to take up to 12 hours on an Intel i7-4790 16 GB machine.

User Defined Features: We include a simple interface for users to extend the considered features in MachSMT's algorithm selection. All that is required is to create a Python method that returns a single floating-point number representing the feature. As input, the user enters the path of the SMT-LIB input, as well as its logic and track. If a user feature is to be considered by MachSMT, the user-defined procedure should return its floating-point representation, otherwise returns none. All user-defined features will automatically be included in building MachSMT. This feature can significantly influence the accuracy of MachSMT when applying it to a specific class of instance.

4 Experimental Results and Analysis

In this section, we present the evaluation of our MachSMT tool (refer to Table 1 and cactus plots in Figures 1–4), specifically with the benchmarks, solvers, and solver runtime analysis from SMT-COMP 2019. More precisely, we used the data from the SMT-COMP 2019 competition. In the SMT-COMP 2019, all solver input queries were performed on the StarExec computing service [43], which consists of a cluster of 2.4 GHz Intel Xeon machines running Red Hat

Logic and Track	**Best Standalone Solver by PAR-2**	**Number of Solved Instances**	**Number Solved by MachSMT**	**PAR-2 % Improvement**
QF_UFBV SQ	Yices 2.6.2	220	222	242.3
NRA SQ	vampire-4.4	82	89	157.1
QF_NRA SQ	Yices 2.6.2	2165	2510	101.8
QF_AUFBV SQ	Yices 2.6.2	38	40	74.2
QF_BV MV	Boolector	7171	7180	73.7
QF_UFBV UC	z3-4.7.1	298	299	66.8
LIA UC	CVC4	200	200	65.4
UFNIA UC	z3-4.8.4	1407	1477	65.0
UFIDL UC	z3-4.8.4	30	30	60.8
BV SQ	Q3B	741	771	58.2
QF_AUFNIA SQ	mathsat	9	9	55.1
QF_LRA SQ	SPASS-SATT	519	517	46.5
AUFNIRA UC	z3-4.8.4	513	517	40.3
QF_LRA UC	CVC4	191	187	39.8
QF_LIA SQ	SPASS-SATT	3048	3069	36.0
QF_AUFLIA UC	Yices 2.6.2	300	300	35.0
QF_UFBV INC	Boolector	1165	1165	31.7
QF_UFLRA INC	z3-4.7.4	1529	1529	29.7
QF_AUFLIA SQ	Yices 2.6.2	651	651	25.8
QF_FP SQ	colibri 2176	196	206	20.8
UFLIA UC	CVC4	3714	3735	16.0
AUFLIRA SQ	z3-4.8.4	1600	1611	15.1
BV INC	z3	10	11	13.2
QF_ANIA SQ	CVC4	7	7	12.3
QF_BV UC	Yices 2.6.0	2437	2459	10.1
QF_UFNRA SQ	Yices 2.6.0	25	23	9.8
QF_ABV UC	Yices 2.6.0	1852	1853	9.3
QF_IDL SQ	z3-4.8.4	938	944	8.4
AUFLIRA UC	z3-4.8.4	9872	9873	8.3
QF_UF SQ	Yices 2.6.2	3512	3512	7.6
AUFLIA UC	CVC4	1161	1166	6.9
BV UC	CVC4	228	228	6.1

Table 1: The 32 (out of 49) SMT-COMP logics on which MachSMT improves over the best standalone solver (SMT-COMP 2019 version) according to the PAR-2 score by more than 5%.

Enterprise Linux 7.2. Each solver/benchmark pair was configured to have 4 cores and 60GB of memory available [22]. Further, we perform k-fold cross validation (with $k = 5$). In cross validation, the dataset is randomly partitioned into k subsets. A model is then trained over $k-1$ subsets and makes predictions over the subset that is excluded from training. This process is repeated to obtain fair predictions for each subset. Cross validation is commonly deployed to analyze machine learning models. For more details we refer to Picard et al. [37].

For every track and logic, we evaluated our MachSMT model by checking whether we beat the best solver from the SMT-COMP 2019 competition, according to PAR-2 scores (refer to results in Table 1). Another metric to evaluate the prediction accuracy of MachSMT is to check how close it is to the virtual best solver in performance. As we discuss below, MachSMT performed better than the best solver from SMT-COMP 2019 competition (PAR-2 score) in 49 out of 102 tracks from the SMT-COMP 2019, with up to a 240% improvement

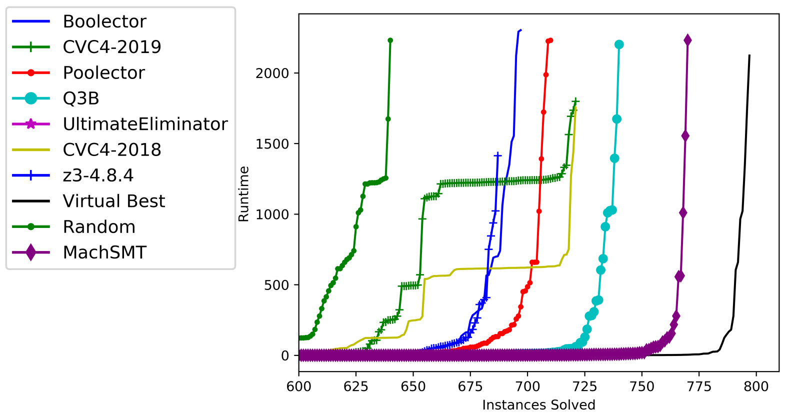

Fig. 1 *Cactus Plot for BV in the Single Query Track.*

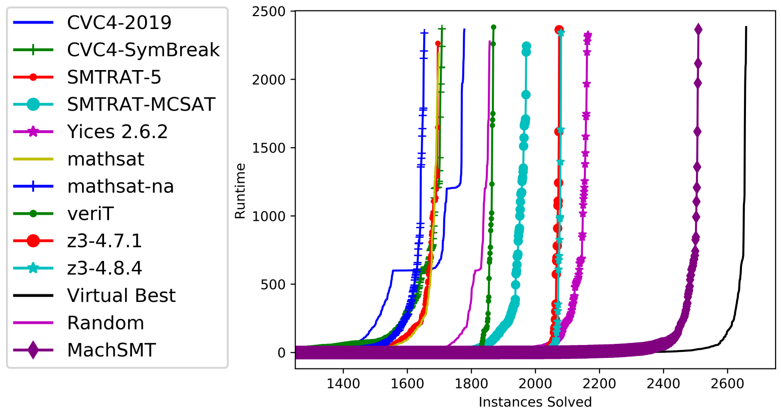

Fig. 2 *Cactus Plot for QF_NRA in the Single Query Track.*

for the QF_UFBV SQ logic. (Certain tracks which have less than 5 benchmarks were excluded from our evaluation since it is very difficult to learn anything from such a small sample.)

We present cactus plots of selected logics and tracks where MachSMT gave the best results[3] in Figures 1–4. A cactus plot is a visualization of how a solver performs on a database of inputs. A point (X,Y) denotes that a solver S solves X inputs within Y seconds each. We additionally include the performance of the virtual best solver (i.e, perfect algorithm selection), random algorithm selection (selects algorithms uniformly-at-random), and all standalone solvers (that competed in the relevant logic) as baselines for comparison.

We use a modified version of the PAR-2 scoring scheme, frequently seen in SAT contests [33]. PAR-2 is the cumulative runtime of a solver over all instances

[3] All cactus plots, results, code can be found on the MachSMT website.

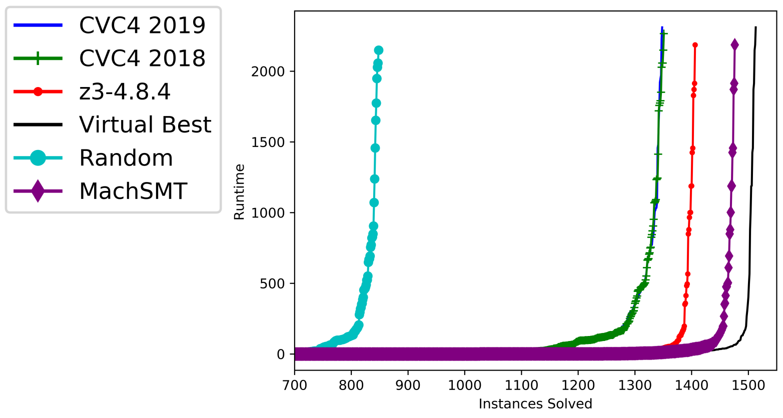

Fig. 3 *Cactus Plot for UFNIA in the Unsat Core Track.*

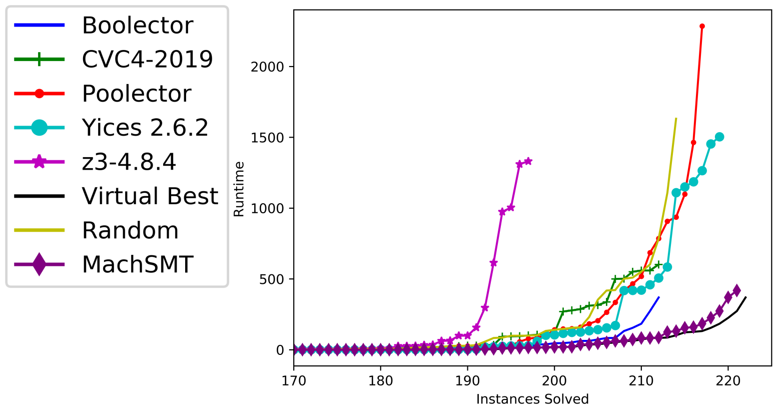

Fig. 4 *Cactus Plot for QF_UFBV in the Single Query Track.*

in the database. On a timeout, a score of twice the wall clock timeout is used. For SAT contests, incorrect answers automatically disqualify a solver, while SMT-COMP penalizes solvers that give incorrect answers. To this end, a wrong answer results in ten times the wall clock timeout in the PAR-2 score. Using MachSMT, we were able to improve upon standalone solvers (i.e., the PAR-2 score winner) in 49 logics across SMT-COMP 2019. We present 32 out of 49 tracks/logics over which MachSMT improves by more than 5% over the best solver according to PAR-2 in Table 1. Further, we compared MachSMT against the PAR4 portfolio solver, which won several logics in the SMT-COMP 2019. We improve over PAR4 in all but two tracks. However, we don't include PAR4 as one of the solvers as part of MachSMT algorithm selector. The reason is that PAR4 is a portfolio solver that uses other existing SMT solvers (from the SMT-COMP 2018 competition) all in parallel, one per thread. The answer produced by the first solver that terminates is returned as the answer by PAR4.

MachSMT is designed to predict the solver with the fastest runtime and to minimize the PAR-2 score. However, we also observed in some logics/tracks that MachSMT gave better PAR-2 (and hence better overall runtime) but solved fewer instances than the corresponding best standalone solver according to PAR-2. We used a total 102 tracks from SMT-COMP 2019 for our evaluation, of which for 10 tracks the best standalone solver was the virtual best solver. Disregarding these, MachSMT improves on the competition winner in 53% of tracks.

5 Related Work

Algorithm selection tools have a rich history and have been around since at least 1976 when Rice et al. were the first to propose it [39]. Algorithm selectors have been extensively used in many contexts, e.g., classifiers for machine learning [1], combinatorics [28], and other NP-hard optimization problems [44, 45]. Within the context of solvers, algorithm selectors have been proposed for QBF [38, 32], SAT [48–50], and CSP solvers [20, 2, 25]. Further, Symbolic Execution tools using SMT solvers have considered algorithm selection [46] for the specific classes of instances within the context of the bit-vector theory. This would be an ideal use case of MachSMT, since we provide a more complete solution. There have been other works using machine learning as applied to SMT solvers to improve their performance. Balunovic et al. use neural networks and synthesis to find tactics and strategies for three SMT-LIB theories [4]. Scott et al. proposed an algorithm selection tool for the QF_FP theory [42]. To the best of our knowledge, MachSMT is the first publicly available tool for the entirety of SMT-LIB.

6 Conclusions and Future Work

In this paper, we present MachSMT, the first algorithm selection tool for the entirety of SMT-LIB. MachSMT is designed to be user-friendly and easily modifiable by users for their specific application. It is not intended as a replacement for any specific SMT solver, but rather aimed at enabling users to leverage the diversity of decision procedures available to them. Using MachSMT, we observe improvement in 49 out of 102 logics in all tracks from the SMT-COMP 2019, with up to a 240% improvement for the QF_UFBV SQ logic. Several of the logics on which we don't see improvement are ones that have very few instances, or are in tracks where our current scoring scheme does not take the specific nature of the track into account. In the future, we plan to improve our scoring scheme for these tracks of SMT-COMP. We further plan to extend our feature set with more (theory-)specific features such as number of quantifier alternations or nesting depth and arity of functions/predicates. It is very likely that users may have domain-specific knowledge about which features might be most predictive of solver runtime. Hence, we have provided an interface to easily extend and specialize MachSMT to a user's specific setting.

References

1. Ali, S., Smith, K.A.: On learning algorithm selection for classification. Applied Soft Computing **6**(2), 119–138 (2006)
2. Amadini, R., Gabbrielli, M., Mauro, J.: Sunny: a lazy portfolio approach for constraint solving. Theory and Practice of Logic Programming **14**(4-5), 509–524 (2014)
3. Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K.S., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for AWS access policies using SMT. In: Bjørner, N., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018. pp. 1–9. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8602994, https://doi.org/10.23919/FMCAD.2018.8602994
4. Balunovic, M., Bielik, P., Vechev, M.: Learning to solve smt formulas. In: Advances in Neural Information Processing Systems. pp. 10317–10328 (2018)
5. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovi'c, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11). Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (Jul 2011), http://www.cs.stanford.edu/ barrett/pubs/BCD+11.pdf, snowbird, Utah
6. Barrett, C., De Moura, L., Stump, A.: Smt-comp: Satisfiability modulo theories competition. In: International Conference on Computer Aided Verification. pp. 20–23. Springer (2005)
7. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2016)
8. Bhargavan, K., Bond, B., Delignat-Lavaud, A., Fournet, C., Hawblitzel, C., Hritcu, C., Ishtiaq, S., Kohlweiss, M., Leino, R., Lorch, J., et al.: Everest: Towards a verified, drop-in replacement of https. In: 2nd Summit on Advances in Programming Languages (SNAPL 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017)
9. Brain, M., D'silva, V., Griggio, A., Haller, L., Kroening, D.: Deciding floating-point logic with abstract conflict driven clause learning. Formal Methods in System Design **45**(2), 213–245 (2014)
10. Brain, M., Schanda, F., Sun, Y.: Building better bit-blasting for floating-point problems. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 79–98. Springer (2019)
11. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Hanna, Z., Nadel, A., Palti, A., Sebastiani, R.: A lazy and layered smt bv solver for hard industrial verification problems. In: International Conference on Computer Aided Verification. pp. 547–560. Springer (2007)
12. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: automatically generating inputs of death. ACM Transactions on Information and System Security (TISSEC) **12**(2), 10 (2008)
13. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathsat5 smt solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 93–107. Springer (2013)
14. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)

15. Drucker, H.: Improving regressors using boosting techniques. In: ICML. vol. 97, pp. 107–115 (1997)
16. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 737–744. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_49, https://doi.org/10.1007/978-3-319-08867-9_49
17. Fu, Z., Su, Z.: Xsat: a fast floating-point satisfiability solver. In: International Conference on Computer Aided Verification. pp. 187–209. Springer (2016)
18. Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: An industrial-strength C model checker. In: 33^{rd} ACM/IEEE Int. Conf. on Automated Software Engineering (ASE'18). pp. 888–891. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3238147.3240481
19. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: International Conference on Computer Aided Verification. pp. 519–531. Springer (2007)
20. Gent, I.P., Jefferson, C., Kotthoff, L., Miguel, I., Moore, N.C., Nightingale, P., Petrie, K.E.: Learning when to use lazy learning in constraint solving. In: ECAI. pp. 873–878. Citeseer (2010)
21. Godefroid, P., Levin, M.Y., Molnar, D.A.: SAGE: whitebox fuzzing for security testing. Commun. ACM **55**(3), 40–44 (2012). https://doi.org/10.1145/2093548.2093564, https://doi.org/10.1145/2093548.2093564
22. Hadarean, L., Hyvärinen, A., Niemetz, A., Reger, G.: Smt-comp 2019. https://www.smt-comp.org/2019, year = 2019
23. Halko, N., Martinsson, P.G., Tropp, J.A.: Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions (2009)
24. Hoerl, A.E., Kennard, R.W.: Ridge regression: biased estimation for nonorthogonal problems. Technometrics **42**(1), 80–86 (2000)
25. Hurley, B., Kotthoff, L., Malitsky, Y., O'Sullivan, B.: Proteus: A hierarchical portfolio of solvers and transformations. In: International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems. pp. 301–317. Springer (2014)
26. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient smt solver for verifying deep neural networks. In: International Conference on Computer Aided Verification. pp. 97–117. Springer (2017)
27. Khadra, M.A.B., Stoffel, D., Kunz, W.: gosat: floating-point satisfiability as global optimization. In: 2017 Formal Methods in Computer Aided Design (FMCAD). pp. 11–14. IEEE (2017)
28. Kotthoff, L.: Algorithm selection for combinatorial search problems: A survey. In: Data Mining and Constraint Programming, pp. 149–190. Springer (2016)
29. Le Goues, C., Leino, K.R.M., Moskal, M.: The boogie verification debugger (tool paper). In: International Conference on Software Engineering and Formal Methods. pp. 407–414. Springer (2011)
30. Leino, K.R.M.: Automating theorem proving with smt. In: International Conference on Interactive Theorem Proving. pp. 2–16. Springer (2013)
31. Leyton-Brown, K., Nudelman, E., Shoham, Y.: Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In: International Conference on Principles and Practice of Constraint Programming. pp. 556–572. Springer (2002)

32. Malitsky, Y.: Evolving instance-specific algorithm configuration. In: Instance-Specific Algorithm Configuration, pp. 93–105. Springer (2014)
33. Marijn Heule, Matti Järvisalo, M.S.: Sat race 2019 (2019), http://sat-race-2019.ciirc.cvut.cz/
34. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. JSAT **9**, 53–58 (2014), https://satassociation.org/jsat/index.php/jsat/article/view/120
35. Păsăreanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. International journal on software tools for technology transfer **11**(4), 339 (2009)
36. 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)
37. Picard, R.R., Cook, R.D.: Cross-validation of regression models. Journal of the American Statistical Association **79**(387), 575–583 (1984)
38. Pulina, L., Tacchella, A.: A multi-engine solver for quantified boolean formulas. In: International Conference on Principles and Practice of Constraint Programming. pp. 574–589. Springer (2007)
39. Rice, J.R., et al.: The algorithm selection problem. Advances in computers **15**(65-118), 5 (1976)
40. Salvia, R., Titolo, L., Feliú, M.A., Moscato, M.M., Muñoz, C.A., Rakamarić, Z.: A mixed real and floating-point solver. In: NASA Formal Methods Symposium. pp. 363–370. Springer (2019)
41. Scott, J., Panju, M., Ganesh, V.: Lgml: Logic guided machine learning. In: AAAI. pp. 13909–13910 (2020)
42. Scott, J., Poupart, P., Ganesh, V.: An algorithm selection approach for qf fp solvers. In: 17th International Workshop on Satisfiability Modulo Theories (2019)
43. Stump, A., Sutcliffe, G., Tinelli, C.: Starexec: A cross-community infrastructure for logic solving. In: International joint conference on automated reasoning. pp. 367–373. Springer (2014)
44. Tierney, K., Malitsky, Y.: An algorithm selection benchmark of the container pre-marshalling problem. In: International Conference on Learning and Intelligent Optimization. pp. 17–22. Springer (2015)
45. Vallati, M., Chrpa, L., Kitchin, D.: Portfolio-based planning: State of the art, common practice and open challenges. AI Communications **28**(4), 717–733 (2015)
46. Wen, S.H., Mow, W.L., Chen, W.N., Wang, C.Y., Hsiao, H.C.: Enhancing symbolic execution by machine learning based solver selection (2019)
47. Wold, S., Esbensen, K., Geladi, P.: Principal component analysis. Chemometrics and intelligent laboratory systems **2**(1-3), 37–52 (1987)
48. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Satzilla-07: the design and analysis of an algorithm portfolio for sat. In: International Conference on Principles and Practice of Constraint Programming. pp. 712–727. Springer (2007)
49. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Satzilla: portfolio-based algorithm selection for sat. Journal of artificial intelligence research **32**, 565–606 (2008)
50. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Satzilla2009: an automatic algorithm portfolio for sat. SAT **4**, 53–55 (2009)