A Machine Learning based Splitting Heuristic for Divide-and-Conquer Solvers

Saeed Nejati[1], Ludovic Le Frioux[2], and Vijay Ganesh[1]

[1] University of Waterloo, Waterloo, ON, Canada
{snejati, vganesh}@uwaterloo.ca
[2] LRDE, EPITA, Le Kremlin-Bicêtre, France
ludovic@lrde.epita.fr

Abstract. In this paper, we present a machine learning based *splitting heuristic* for divide-and-conquer parallel Boolean SAT solvers. Splitting heuristics, whether they are look-ahead or look-back, are designed using *proxy metrics*, which when optimized, approximate the true metric of minimizing solver runtime on sub-formulas resulting from a *split*. The rationale for such metrics is that they have been empirically shown to be excellent proxies for runtime of solvers, in addition to being cheap to compute in an online fashion. However, the design of traditional splitting methods are often ad-hoc and do not leverage the copious amounts of data that solvers generate.

To address the above-mentioned issues, we propose a machine learning based splitting heuristic that leverages the features of input formulas and data generated during the run of a divide-and-conquer (DC) parallel solver. More precisely, we reformulate the splitting problem as a ranking problem and develop two machine learning models for *pairwise ranking* and computing the *minimum ranked* variable. Our model can compare variables according to their *splitting quality*, which is based on a set of features extracted from structural properties of the input formula, as well as dynamic *probing statistics*, collected during the solver's run. We derive the true labels through offline collection of runtimes of a parallel DC solver on sample formulas and variables within them. At each splitting point, we generate a predicted ranking (pairwise or minimum rank) of candidate variables and split the formula on the top variable. We implemented our heuristic in the Painless parallel SAT framework and evaluated our solver on a set of cryptographic instances encoding the SHA-1 preimage as well as SAT competition 2018 and 2019 benchmarks. We solve significantly more instances compared to the baseline Painless solver and outperform top divide-and-conquer solvers from recent SAT competitions, such as Treengeling. Furthermore, we are much faster than these top solvers on cryptographic benchmarks.

1 Introduction

Boolean satisfiability (SAT) solvers are powerful general purpose search tools that have had a revolutionary impact on many different domains, such as software engineering [9], AI [37], and cryptography [30, 32]. They get their power

from proof-construction components like clause learning [29] and heuristics aimed at optimally sequencing, selecting, and initializing proof rules such as branching [31, 22] and restarts [3].

The availability of many-core machines has led to a considerable effort in parallel SAT solver research in recent years [5]. Broadly speaking, researchers have developed two parallel SAT solver strategies, namely, *portfolio* and *divide-and-conquer* (DC) solvers. A portfolio SAT solver consists of a set of sequential worker solvers, each implementing a different collection of heuristics, and all of them attempting to solve the same instance running on different cores of a many-core machine. The key principle behind a portfolio solver is that of the *diversity of heuristics*, i.e., by leveraging a diverse set of heuristics to solve an instance may be more efficient than just using a single heuristic given the well-known fact that different classes of formulas are often best solved by distinct methods [11]. On the other hand, DC solvers partition the search space of the input formula and solve each sub-formula using a separate sequential worker solver. Each sub-formula is a restriction of the input formula with a set of assumptions [41]. In both the portfolio and DC settings, the sequential worker solvers may share clauses to exchange useful information they learn about their respective search spaces.

In the context of DC solvers, a splitting heuristic is a method aimed at choosing the "next variable" to add to the current list of assumptions (also known as guiding paths [41]). A bit more formally, one can define a splitting heuristic as a function that takes as input features of a given formula ϕ and/or statistics of a DC solver's state and outputs a variable to split on. Splitting heuristics are typically dynamic, i.e., they re-rank variables at regular intervals throughout the run of a DC solver.

The process of splitting itself can be described as follows: for a given input formula ϕ, say that a variable v is chosen for splitting. The solver generates $\phi[v = F]$ (resp. $\phi[v = T]$) by setting v to False (resp. True) and appropriately simplifying the resultant sub-formulas using Boolean constraint propagation (BCP). These two sub-formulas are then solved in parallel. Each of these sub-formulas can be further split into smaller sub-formulas recursively. Many heuristics for splitting have been studied in the literature [17, 2, 1, 35].

Splitting heuristics can be broadly categorized as *look-ahead* and *look-back*. Look-ahead heuristics choose some subset of variables in the input formula, analyze the impact of splitting on these variables, and rank them based on some measure that correlates well with minimizing runtime[3] of the solver on the sub-formulas thus obtained. By contrast, look-back heuristics compute statistics on "how well a variable participated in the search exploration in the past" (e.g., in clause learning, propagation, etc.), rank them appropriately, and split on the highest-ranked variable. Examples of look-back heuristics include splitters based on VSIDS activity [2], number of flips [21], and propagation-rate [35].

While considerable work has been done on splitting heuristics, almost all previous approaches share the following characteristics: they compute some fea-

[3] Runtime of a solver here refers to the wallclock time of solving a formula.

tures of the input formula and/or statistics over the solver state at appropriate intervals during the solver's run, and then use these as input to a "hand-coded" function (a splitting heuristic designed by the solver designer), that in turn computes a metric correlated with solver runtime to pick the "best" variable to split. By metric we mean a quantity that can be used to rank variables of the input formula such that splitting on the highest-ranked variable ideally corresponds to minimizing solver runtime. We argue that the design of splitting heuristics can be dramatically improved by leveraging a data-driven machine learning (ML) approach, especially for families of formulas (e.g., cryptographic instances) where it can be hard for human designers to come up with effective "hand-coded" splitting heuristic.

In this paper, we propose two ML-based methods, namely *pairwise ranking*, and *min-rank*. The pairwise ranking model takes as input features of a given formula ϕ, aspects of solver state, as well as features of a pair of variables v and u, and ranks them in descending order based on some splitting metric. This ML-based "comparator" is in turn used by our DC solver to rank variables for splitting at regular intervals during its run. The min-rank model, takes as input features of a given formula ϕ, aspects of solver state, and features of a variable v, and outputs whether the variable v has the minimum rank among all variables of the input formula (i.e is it the best variable to split?). Both of these models are binary classifiers implemented using *random forest*.

We implemented our heuristics in the Painless parallel solver framework [20] (we refer to our solver as MaplePainless-DC), and compared it with top parallel SAT solvers from recent SAT competitions. We find that our ML-based method out-performs the best DC solvers on both SAT 2018/2019 competition as well as cryptographic instances.[4]

Contributions. In greater detail, our main contributions are as follows:

1. **MaplePainless-DC: A DC Solver based on ML-based Splitters.** We present MaplePainless-DC, an ML-based splitting DC parallel SAT solver. To the best of our knowledge, MaplePainless-DC is the first parallel solver with an ML-based splitting heuristic. Briefly, our splitting heuristics are ML models, trained offline on both static formula/variable features (e.g., variable occurrence in binary clauses) as well as "dynamic" features based on aspects of the solver's state at runtime (e.g., number of times a variable has been assigned, activities). We propose and implement two different models, namely, *pairwise ranking* and *min-rank*, described above. At runtime, the trained ML-model is invoked by MaplePainless-DC on a vector of static and dynamic variable features at appropriate intervals, which in turn outputs a ranking of the variables in the input formula. The splitting heuristic then chooses the top-ranked variable, splits the formula by assigning that variable both True and False values, and gives the resultant sub-formulas to worker solvers to solve. (See Section 3)

[4] We only compare our MaplePainless-DC solver against the state-of-the-art DC solvers because it is well-known that the most notable portfolio solvers often outperform the DC solvers on application benchmarks.

2. **Evaluation on Cryptographic Instances.** We evaluated our splitting heuristics on a cryptographic benchmark of 60 instances encoding preimage attack on round-reduced SHA-1 function (inversion of 60 random hash targets). We used top sequential solvers in solving cryptographic instances as backend solvers (MapleSAT and Glucose). We outperform the baseline solver (Painless-DC with the same backends and flip as splitting heuristics) in an apple-to-apple comparison, solving an additional instance from the hardest subset of instances and 30% faster on average on solved instances. We also solve 19 more instances (over a benchmark of 60) and are significantly faster relative to one of the top DC solvers, Treengeling. (See Section 5.2)
3. **Evaluation on SAT Application Instances from SAT 2018 competition and SAT 2019 race.** We evaluated our splitting heuristics on main track benchmarks of SAT competition 2018 and SAT race 2019 (total 800 instances) against the baseline solver (Painless-DC with flip as splitting heuristic) in an apple-to-apple comparison, as well as against Treengeling. On the combined SAT 2018 and SAT 2019 benchmarks, we outperform both these solvers in terms of the number of solved instances and PAR-2 score[5]. Furthermore, MaplePainless-DC solves satisfiable instances much better than all other solvers (18 more than both the baseline and Treengeling solvers overall application instances), when using the pairwise ranking model. (See Section 5.1)

2 Background

In this section, we list relevant definitions and notations. We refer the reader to [7] for details on CDCL SAT solvers. By the term "split" or "splitting" a formula ϕ over variable v we refer to the process of generating two sub-formulas $\phi_1 = \phi \wedge \neg v$ and $\phi_2 = \phi \wedge v$, which are assumed to be simplified via unit or Boolean constraint propagation.

DC solvers take as input a Boolean formula and split it into many smaller sub-formulas, solve them using sequential worker solvers, and combine the results (SAT if at least one sub-formula is SAT, UNSAT if all of them are UNSAT). The architecture is usually of a master-slave type, where the slaves are sequential solvers and the master node maintains the splittings in the form of a search tree. Each node of the tree is a variable and branches correspond to setting that variable to True or False. Each "root to leaf" path represents a set of assumptions, also known as guiding path or cube. The phrase "solving a cube" refers to solving the original formula constrained with the given cube.

The notation $t_S(\phi)$ refers to the time to solve a Boolean formula ϕ with a sequential worker CDCL SAT solver S (We drop the subscript if it is clear from context). We denote the reduced formula after setting v to False (respectively to True) with $\phi[v = F]$ (respectively, $\phi[v = T]$). By reducing a formula we mean

[5] PAR-k is the Penalized Average Runtime, counting each timeout as k times the wallclock timeout.

simplification via unit propagation (i.e., removal of satisfied clauses from the formula, falsified literals from clauses).

The term *performance metric*, with respect to a given solver S, refers to a function $pm : \phi \times v \to \mathbb{R}$, over a formula ϕ and a variable $v \in vars(\phi)$, that characterizes the "quality" of splitting ϕ over v. Minimizing this metric ideally should correlate with minimizing solver runtime.

More precisely, the general goal of designing a splitting heuristic is twofold: first, to come up with a metric that correlates with minimizing solver runtime, and second to design a function to compute said metric. Researchers have proposed a variety of performance metrics in the context of splitting heuristics. Below are definitions of three such performance metrics and the intuition behind each of them. In previous work, researchers have found that these metrics are good proxies for minimizing runtime in the context of splitting in DC solvers. Further, to state the obvious, it is ideal to split on a variable that minimizes these metrics over all variables of an input formula. Let $\phi_1 = \phi[v = F]$ and $\phi_2 = \phi[v = T]$, be the sub-formulas after splitting ϕ over v.

- $pm_1(\phi, v) = \max\{t(\phi_1), t(\phi_2)\}$: This metric aims to capture the runtime of a DC solver executed in parallel over the sub-formulas ϕ_1 and ϕ_2.
- $pm_2(\phi, v) = t(\phi_1) + t(\phi_2)$: This function gives higher priority to splitting variables that make the problem easier even in a single core setting.
- $pm_3(\phi, v) = -(t(\phi) - t(\phi_1)) \cdot (t(\phi) - t(\phi_2))$: The idea behind this metric is to measure runtime "progress" in each branch (by comparing the runtime of sub-formulas with the original formula) and also aims to balance the hardness of the two branches.

Random Forest Classification. We refer the reader to the paper by Liaw et al. on random forests [26]. Briefly, the random forest is an ensemble learning method, that constructs a set of decision trees at training time and outputs the class that appears most often at the output of decision trees. Decision trees are a popular method for various machine learning tasks. However, trees that grow very deep tend to learn highly irregular patterns: they overfit their training sets, i.e. have a low bias, but very high variance. Random forests are a way of averaging multiple deep decision trees, trained on different parts of the same training set, with the goal of reducing the variance.

3 Machine Learning Models for Splitting

In this section we discuss a formulation of the splitting problem, define a quality measure for splitting, and study how we can train ML models that approximate the best splitting variable.

3.1 The Splitting Problem

Given a Boolean formula ϕ, a sequential solver S, and performance metric pm, the splitting problem is to determine a variable v in ϕ such that the time

required to solve each of $\phi[v = T]$ and $\phi[v = F]$ by S is minimal over all variables in ϕ with respect to the given performance metric pm, i.e. to find $\text{argmin}_{v \in vars(\phi)}\{pm(\phi, v)\}$.

Modeling the exact behavior of a DC solver as it solves the sub-formulas in parallel and splits them on demand, is a challenging task. Below we define a metric that we believe is a more accurate measure of the optimal choice of a splitting variable, compared to the heuristic metrics mentioned in Section 2.

Let $\phi_1 = \phi[v = F]$ and $\phi_2 = \phi[v = T]$ be sub-formulas of splitting ϕ over v, and let $t_1 = t_S(\phi_1)$ and $t_2 = t_S(\phi_2)$ be runtimes of solving them by sequential solver S. The total time taken to solve the formula ϕ in this setting depends on the status and runtimes of the sub-formulas. If ϕ is UNSAT, the solver needs to prove both of the sub-formulas UNSAT. Hence the total time to solve such an instance is the maximum of the solver runtimes over the two sub-formulas. If on the other hand the formula ϕ is SAT, at least one of the sub-formulas must be SAT. If both sub-formulas are SAT, the total time is the minimum of the two, otherwise, only the SAT sub-formula matters. The total time of solving ϕ after splitting over variable v can be represented as follows:

$$T_{total}(\phi, v) = \begin{cases} \max(t_1, t_2), & \phi_1 : UNSAT, \phi_2 : UNSAT \\ t_2, & \phi_1 : UNSAT, \phi_2 : SAT \\ t_1, & \phi_1 : SAT, \quad \phi_2 : UNSAT \\ \min(t_1, t_2), & \phi_1 : SAT, \quad \phi_2 : SAT \end{cases}$$

We use this total runtime as our performance metric: $pm(\phi, v) = T_{total}(\phi, v)$. In other words the target of our splitting heuristic is: given formula ϕ, find a variable $v = \text{argmin}_{v \in vars(\phi)}\{T_{total}(\phi, v)\}$.

3.2 Handling Timeouts

In practice, sub-formulas obtained after splitting on a variable can be hard for SAT solvers, and thus they may timeout for those cases. Let the status of a timed out (sub-)formula be labeled as "UNKNOWN". For a pair of variables u and v in formula ϕ, we collect the runtime and status of solving sub-formulas $u_1 = \phi[u = F], u_2 = \phi[u = T], v_1 = \phi[v = F]$ and $v_2 = \phi[v = T]$. If the status of all four of these sub-formulas is UNKNOWN, we cannot derive the truth label (we do not know which of these two variables is better for splitting). In all other cases (mix of having SAT/UNSAT and UNKNOWN), we have enough information to be able to compare u and v.

3.3 Learning to Rank

Generally, performance metrics can be used to generate a total order over the splitting variables (the higher ranked variables have a higher performance metric). Thus we can see the splitting problem as picking the minimum element from a ranked list. A common way of implementing splitting heuristics is to rank

the variables by directly deriving the performance metric of each variable and selecting the minimum element. However, this is not the only way one can rank the elements. There are three main approaches in the ML literature for learning a model to rank a list of elements [28]:

- **Pointwise**: Learning a numerical or ordinal score for each data point, which are in turn sorted according to their ordinal score. The problem here translates to training a regression model.
- **Pairwise**: In this approach, ranking is done via learning a model that acts as a comparator, which takes as input two data points and outputs a total order over them.
- **Listwise**: These algorithms try to directly minimize a ranking evaluation metric (e.g. τ-score or Mean Average-Precision) that compares a predicted ranking against a true ranking.

Almost all previous branching and splitting heuristics use pointwise ranking. For example, VSIDS branching heuristics [31] maintains a score for each variable, which represents how much that variable participated in clause learning recently. Then the variable with the highest activity is picked. Ultimately, the goal is to minimize the runtime and one might learn a function that directly approximates the desired runtime based ranking. However, approximating the runtime distribution of the CDCL SAT solver is very hard in general, as the interplay of the many heuristics in CDCL solvers makes it hard to predict how the search progresses. Heuristic designers hope that their variable ranking strongly correlates with a ranking where high ranking variables generate easier sub-formulas. In other words, their variable ranking using the proxy metric strongly correlates with runtime-based ranking. In the case of splitting or branching heuristics, we do not care about the actual runtime of sub-formulas and only want to know which variable corresponds to the lowest runtime. In other words, we want a way of comparing runtimes and not exactly deriving the runtime values. As mentioned above, we are looking for a minimum element in an array, sorted based on a metric. We approach this task of finding the minimum using two different methods. First, we build a *pairwise ranking* model that learns to compare two elements (two variables in our case), and second, we use a modified version of ordinal ranking, that we call *min-rank*, where we build a classifier that determines whether a given variable sits at the rank 1. We used binary classification for building both of these models. In the pairwise ranking, we use the model as a less-than operator and find the minimum in a linear scan. In min-rank, we check all of the variables against the model and pick the variable that the model declares as the minimum.

The first model is represented by a binary classifier PW (PairWise) that takes as input features of a formula ϕ and features of two variables v_i and v_j within ϕ, and answers the question of "is v_i better than v_j for splitting ϕ?" (according to our splitting performance metric described in Section 3.1).

$$PW(\phi, v_i, v_j) = \begin{cases} 1, & pm(\phi, v_i) < pm(\phi, v_j) \\ 0, & otherwise \end{cases} \tag{1}$$

This type of predicate learning was also used in one of the SATZilla versions [40] (known as pairwise voting), to rank a list of algorithms on a given instance.

For the second model, we used the idea of reduction by Lin et al. [27] for implementing ordinal ranking using binary classification. In their work, the role of a binary classifier given an element and an integer rank k is to determine whether the element is within the top k elements or not. Splitting heuristics look for the top variable in a ranked list, thus we are only interested in the $k = 1$ case. We define a binary classifier MR (Min-Rank) that takes as input a variable v, and answers the question "is v the best variable for splitting ϕ?".

$$MR(\phi, v_i) = \begin{cases} 1, & \forall j \neq i : pm(\phi, v_i) < pm(\phi, v_j) \\ 0, & otherwise \end{cases} \tag{2}$$

3.4 Features for Training the Models

The data points that we used to train the model have the following format:

$$\begin{aligned} PW &: (\langle formula_{features}(\phi), var_{features}(v_i), var_{features}(v_j)\rangle, \{0, 1\}) \\ MR &: (\langle formula_{features}(\phi), var_{features}(v)\rangle, \{0, 1\}) \end{aligned} \tag{3}$$

where the last element corresponds to the appropriate classifier ($PW(\phi, v_i, v_j)$ or $MR(\phi, v)$). For the formula features, we started from the features proposed by SATZilla in SAT competition 2012 [40]. Compared to the model that has been used in SATZilla, we will query our model at each splitting point. The feature computation time can quickly become a big part of the total runtime, and dominate the gain from picking a better splitting variable. On the other hand, each of the features could have an important role in making the model representative of the target distribution. To address this problem we performed a feature selection on our initial set of features (both formula and variable features). We first removed the very heavy features like LP-based (linear programming) features. We used the random forest for training our models. We then extracted the relative importance of each feature after training, which corresponds to the frequency of the appearance of those features in the ensemble of decision trees. We created a sorted list of features based on their relative importance (f) and performed a forward feature selection [10]. More specifically, starting with an empty list F, we passed through f and added the features to F, if they reduced the cross-validation error when training on F. We then performed a backward pass on F, to remove heavy-to-compute features (having normalized cost of at least 100 milliseconds), that do not contribute much to the accuracy of the model (having feature importance in the 25th percentile). We also took into account the product features (features from the multiplication of pairs of other features) to add non-linearity to the model. The final variable and formula features are listed in Table 1, consisting of structural metrics and metrics from a limited search. The features are listed in order of their importance extracted from the trained random forest.

Table 1: Variable ($var_{features}(v)$) and Formula features ($formula_{features}(\phi)$), sorted based on their importance extracted from the trained models.

Feature name	Description
numAssigned	#times v got a value through branching/propagation
numFlip	#times the implied value of v is different than its cached value [1]
numLearnt	#times v appeared in a conflict clause
numInTernary	#times v appears in a clause of size 3
numInBinary	#times v appears in a clause of size 2
LRBProduct	product of LRB [22] activities of v and $\neg v$ literals
propRate	average #propagation over #decision [35]
activity	VSIDS activity [31]
numPropagations	number of unit propagations in the limited search
conflictRate	ratio of #conflict clauses over #decisions
totalReward	sum of LRB reward of all of the variables
numBinary	number of clauses of size 2 in ϕ
numTernary	number of clauses of size 3 in ϕ
avgVarDegree	average variable node degree in the Variable-Clause graph
avgClauseDegree	average clause node degree in the Variable-Clause graph

3.5 Training Data

We used the MapleCOMSPS solver [23] for collection of solver runtime, as well as formula and variable features. For generating our training data set, we picked 210 instances from the collection of application/crafted benchmarks of SAT competition 2016 [14] and 2017 [15]. To be more precise, 87 instances from the application benchmark of 2016, 21 instances from the crafted benchmark of 2016, and 102 instances from the main benchmark of 2017. The selection criteria were based on having instances from different types of problems (not problems of the same kind with different sizes) and having a wide range of hardness to make a representative training set. We did not use any instance that was deemed too hard (timed out) or too easy (was solved under 5 seconds) by our sequential solver. To match the test environment, we first ran the pre-processing stage of MapleCOMSPS and simplified the formulas. Then we computed all of the structural formula features offline and for the search probing features we ran MapleCOMSPS up to 10,000 conflicts and collected the necessary statistics from the solver. For computing the true labels, we randomly selected 50 variables in each instance and split the formula on each of them and solved the sub-formulas with MapleCOMSPS up to a 5000 seconds timeout, recording the runtime and status (SAT, UNSAT, UNKNOWN).

3.6 Analysis of the Learned Models

For training the model, we used *random forest classifier*. We can achieve an average precision of 83% and an average recall of 83% and an accuracy of 80.7%. The candidate variable list can be ordered using the learned predicate. For finding the best variable, we only need to find the "min" of the list, which can be done

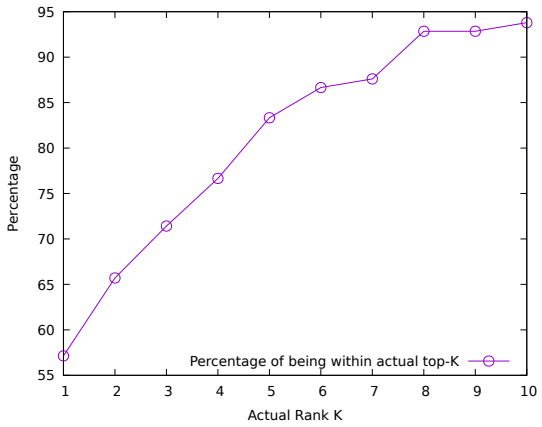


Fig. 1: Percentage of instances where the predicted best variable is within the actual top-k variables for k between 1 and 10.

in linear time. Although, when using a noisy comparator, the error caused by the inaccurate comparison, might accumulate over multiple comparisons. There are more robust sorting algorithms in the presence of noisy comparators (e.g. counting method [38]), but the running time complexity is quadratic in the number of elements, which is not feasible for large formulas. To check how well our predicate is performing, we ranked the variables in the instances in our training set, where we have the true labels.

When sorting the variables using the pairwise ranking, out of 210 instances, in 120 instances the best variable in the predicted ranking matched the actual best variable (57.1% of the time). In 18 cases the best actual variable was the second best predicted variable. The worst prediction happened in an instance with 2200 variables, where the best actual variable appeared in the 30th position in the predicted list. The total error (e.g. τ score) of comparing the predicted ranking and the best actual ranking could be poor, however, we can see some general ordering over the variables (variables that are much better choices appear closer to the front of the list). Figure 1 shows the percentage of instances (out of 210), where the predicted best variable (the output of the model for splitting), is within the actual top-k variables. We observed that the best predicted variable is one of the actual top 10 variables in 197 out of 210 instances (93.8%). This shows that top variables in our predicted ranking have a considerable overlap with the top variables in the actual ranking.

4 Implementation

Our implementation of MaplePainless-DC is built on top of the Painless solver framework [20]. Painless is a state-of-the-art framework that allows developers to implement many different kinds of parallel SAT solvers for many-

core environments. The main configurable components of Painless are: parallel strategies such as DC or portfolio, clause sharing, and management policies, and diverse sequential engines. The implementation of our machine learning based splitting heuristic relies on the use of the DC strategy in Painless [21]. We use an instrumented version of the MapleCOMSPS [23] solver as workers in MaplePainless-DC. The instrumentation collects formula/variable statistics and chooses splitting variables.

4.1 Implementation of Splitting in Painless-DC

Painless-DC splits a formula at regular intervals throughout its run. At a high-level, the master node maintains a queue of idle cores to assign jobs to. Initially, the master node chooses a variable to split and assigns the resultant sub-formulas to two cores. If the queue of idle cores is non-empty, the master node chooses a sub-formula from one of the busy cores and splits it into two sub-formulas, one of which is assigned to the busy core and the other to one of the idle ones. This process is repeated until the queue of idle cores is empty. If during the solver's run a core becomes idle and is added to the idle queue (e.g., if it has established UNSAT for its input sub-formula), the above-mentioned process is invoked until the idle queue becomes empty again. This form of load-balancing ensures that worker nodes are not allowed to idle for too long.

4.2 Feature Computation for Machine Learning in MapleCOMSPS

When it is time to split a formula, Painless' master node asks the sequential worker solver whose sub-formula is being split for variables to split on. The worker solver computes formula and variable features (e.g. number of times a variable is assigned, either decided or propagated) on the sub-formula to be split. The description of the variable features is listed in Table 1.

We used scikit-learn python package [36] for training the model and extracted the parameters and embedded them in a C implementation of random forest classifier. We later call this classifier from MapleCOMSPS for performing predictions. Given a list of candidate variables, pairwise classifier PW is used as a comparator (less-than) operator to find and return the minimum item in a linear scan. Min-rank classifier MR is invoked for all variables in the list and the first variable predicted to be the minimum is returned. The worst case time complexity of both of the models is $O(T_C \cdot n)$, where n is the number of variables and T_C is the time complexity of querying each of the classifiers.

5 Experimental Results

5.1 Evaluation over SAT 2018 and 2019 Competition Instances

Experimental Setup. For evaluation we used the main track benchmark of the SAT competition 2018 [13] and SAT race 2019 [12], which in total have

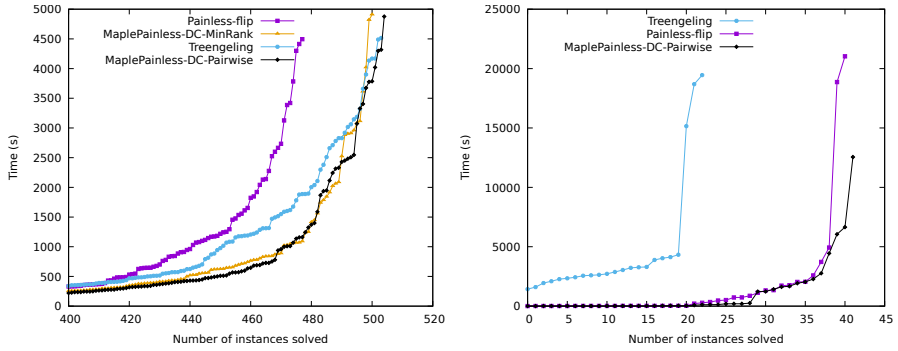
Table 2: Performance comparison of our solvers vs state-of-the-art DC parallel SAT solvers. Number of solved instances is out of 791 (after removing repeated instances from the original 800). SAT column shows the number of satisfiable instances solved (resp. UNSAT). The best result in each column is shown in bold.

Cores	Solver	Solved	SAT	UNSAT	Avg. Runtime (s)	PAR-2 (hr)
8	`Treengeling`	**501**	292	**209**	719.399	905.672
	`Painless-flip`	474	291	183	437.632	938.177
	`MaplePainless-DC-MinRank`	497	299	198	484.340	883.532
	`MaplePainless-DC-Pairwise`	**501**	**309**	192	**435.610**	**866.178**
16	`Treengeling`	518	308	**210**	677.216	855.777
	`MaplePainless-DC-Pairwise`	**520**	**317**	203	**334.991**	**801.165**

800 instances, consisting of industrial instances coming from a diverse set of applications and crafted instances encoding combinatorial problems. Within our sample of instances from 2016/2017 (used for training), a scrambled version (a shuffling of clauses and variable IDs) of 9 instances appear in the 2018/2019 benchmarks as well (used for testing). To have a fair evaluation, we removed these 9 instances from the testing benchmark. Timeout for each instance was set at 5000 seconds wallclock time (the same as in SAT competitions). All jobs were run on Intel Xeon CPUs (3GHz and 64GB RAM). As a sanity check, we performed a controlled apple-to-apple study comparing Painless with ML-based splitting heuristic against the same setup with random splitting heuristic. We note that Painless with ML-based splitting easily outperforms the version with random splitting.

Solvers Description. We compared our solver against the top divide-and-conquer parallel solvers, Treengeling [6] version bcj and Painless-DC [21] with its best performing setting (node switch strategy: *clone*, clause sharing: all-to-all, and splitting heuristic: *flip*), which we will refer to as `Treengeling` and `Painless-flip`, respectively. We refer to our implementations using the PW classifier for pairwise ranking as `MaplePainless-DC-Pairwise` and `MaplePainless-DC-MinRank` refers to solver with binary classification of minimum rank (MR classifier). Our parallel solvers and `Painless-flip` use MapleCOMSPS [23] as the backend sequential solver. We changed MapleCOMSPS to always use LRB as branching heuristics. Each solver was assigned 8 cores.

Results. To perform an apple-to-apple comparison and measure the effectiveness of our splitting heuristics, we reused all of the configurations and components of `Painless-flip` and only replaced the splitting heuristics, which was straightforward, thanks to the modular design of Painless. Table 2 lists the number of solved instances, average runtime among solved instances, and the PAR-2 metric. In the SAT competition, PAR-2 is measured in seconds, but for better readability, we report it in hours. As the table shows, both ML based heuristics,



(a) SAT 2018/2019 benchmark (b) Hard cryptographic benchmark

Fig. 2: Cactus plot for performance comparison of our parallel SAT solvers against baseline and state-of-the-art on main track benchmarks of SAT competition and hard cryptographic benchmark (using 8 cores).

improve significantly upon the baseline in the application benchmark of 2018 and 2019. Additionally, `MaplePainless-DC-Pairwise` solves the same number of instances as `Treengeling`, but has the lowest PAR-2 score among all. Figure 2a shows the cactus plot of these solvers over this benchmark.

5.2 Evaluation over Cryptographic Instances

Experimental Setup. We used a set of hard cryptographic instances encoding preimage attack on round-reduced SHA-1 hash function. More precisely, the instances encode inversion of 21, 22, and 23 rounds SHA-1, with 20 random targets for each rounds version [33]. All jobs were run on Intel Xeon CPUs at 3GHz and 64GB of RAM with 12 hours wallclock timeout.

Solvers Description. We compared our `MaplePainless-DC-Pairwise` solver against the baseline (`Painless-flip`) and `Treengeling`. All solvers were run with 8 cores. For the backend solvers in this experiment, we used Glucose [4] and MapleSAT [22] (4 of each). Glucose solvers used Glucose's default restart policy. MapleSAT solvers were set to use the MABR restart policy [34]. To have an apple-to-apple comparison with baseline, we used the same backend solver configuration for baseline and our solvers.

Results. Figure 2b shows the performance of the considered DC solvers on our hard cryptographic benchmark. Instances with 21 rounds are easy for all solvers. 22 rounds instances are much harder than 21 rounds instances and as can be seen, `Treengeling` solves very few of these instances. Although both `MaplePainless-DC-Pairwise` and `Painless-flip` solve all of these instances.

The hardness ramps up very quickly at 23 rounds instances, where `Treengeling` does not solve any of the instances and `Painless-flip` solves 2 of them. `MaplePainless-DC-Pairwise` solves 3 instances in this subset of instances, and with 30% lower runtime.

5.3 Scaling Experiments

Our main set of experiments were executed on 8 CPU cores. To study how our splitting heuristic scales with larger number of cores, we took `Treengeling` and `MaplePainless-DC-Pairwise`, that performed better among the four solvers on SAT 2018 and 2019 benchmarks, and compared them on these benchmarks on 16 core machines. Table 2 shows that our `MaplePainless-DC-Pairwise` solver can solve 2 more instances than `Treengeling` (as opposed to solving the same number of instances as observed in the 8 core setting). Further, `MaplePainless-DC-Pairwise` with 16 cores solves 19 more instances compared to the same version with 8 cores, and 11 of these instances were unsatisfiable.

5.4 Computational Overhead of ML Models

The timing results presented in this section are end-to-end (i.e., the computational overhead of running the ML models are included in the solver runtimes presented). The majority of the variable features are dynamic and their counters are updated whenever there is a related action performed during the search, thus their complexity is amortized over the run of the solver. The structural formula features are computed at the start of the search, which are all linearly proportional to the size of formula, and later are updated incrementally as the formula is reduced via splitting. Setting up the feature values and querying the models roughly takes 6% of the total runtime of the solver on average for the SAT 2018 and 2019 benchmarks.

5.5 Summary of Results

We first note that MaplePainless-DC significantly outperforms both baseline as well as the state-of-the-art `Treengeling` solvers on cryptographic (60 instances) and SAT 2018/2019 competition benchmarks (800 instances) both in terms of number of solved instances as well as PAR-2 scores. Further, we see an improvement in performance of our solver MaplePainless-DC as we increase the number of machine cores from 8 to 16 (see Table 2).

Both of the ML-based heuristics are very successful on satisfiable instances, where `MaplePainless-DC-Pairwise` solves 17 more satisfiable instances relative to `Treengeling` and 18 relative to `Painless-flip` (although solving fewer unsatisfiable instances than `Treengeling`). On cryptographic benchmark, `MaplePainless-DC-Pairwise` solves 43 out of 60 instances, outperforming other solvers. From the hardest instances (23 rounds SHA-1) in this benchmark, `Treengeling` can not solve any of the instances, whereas `MaplePainless-DC-Pairwise` solves three of them (see Table 2 and Figure 2).

6 Related Work

Cube-and-conquer [16] solvers (such as Treengeling [6]) use a look-ahead procedure to determine the best splitting variable. In contrast to look-ahead techniques, some solvers use look-back methods that dynamically analyze the search performed by the solver, as well as formula statistics, to identify the best candidate at the "current" splitting point. For example, Ampharos [2] picks the variable with the highest VSIDS activity and MapleAmpharos [35] uses propagation-rate (average propagation of a variable divided by the number of decisions). Audemard et al. [1], use the number of times a variable's saved phase is flipped through propagation. This has been shown to be effective in divide-and-conquer settings [21]. We can categorize our work as a look-back heuristic as all of the features are extracted from previous limited runs of a sequential solver.

ML has been used to rank and pick the best variable in sequential SAT solvers. Liang et al. used a reinforcement learning formulation to find the most rewarding variable according to the learning-rate metric for branching [22]. In another work, they train a logistic regression model that ranks variables based on the probability of causing a conflict in the next step [25]. In contrast to these methods that use a pointwise ranking of the variables, we are employing a pairwise ranking. The pairwise ranking has been used in other constraint solver contexts as well. Xu et al. used pairwise voting in the context of algorithm selection, to rank SAT solvers based on their performance on a single formula [40]. Khalil et al. used deep reinforcement learning for learning heuristics in optimization algorithms over graphs of up to 1000 nodes [18], however, there is a scaling challenge when applying their work on industrial SAT instances which can have millions of variables.

7 Conclusions

We presented two ML based look-back splitting heuristics for DC solvers in this paper, namely, pairwise ranking PW and min rank MR methods. These methods significantly outperform the baseline Painless and state-of-the-art `Treengeling` solvers on both industrial and cryptographic benchmarks.

One of the key insights that underpins our solver heuristic design is the observation that solvers are compositions of two kinds of methods, namely, logical reasoning routines (e.g., conflict clause learning or BCP), and heuristics aimed at optimally selecting, sequencing, or initializing logical reasoning rules.We show that our methods outperform hand-tuned heuristics in the best DC solver to-date, namely, `Treengeling`, on a large industrial benchmark as well as challenge problems obtained from cryptographic applications. This gives us greater confidence in our philosophy that design of solver heuristics can effectively leverage ML methods, especially given the fact that solvers are data-rich environments. Further, future solver design is likely to move away from ad-hoc heuristic design and more towards feature engineering and appropriate choice of ML methods, as has already been witnessed for many solver heuristics [22, 25, 24, 8, 39, 19].

References

1. Audemard, G., Hoessen, B., Jabbour, S., Piette, C.: An effective distributed D&C approach for the satisfiability problem. In: Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP). pp. 183–187. IEEE (2014)
2. Audemard, G., Lagniez, J.M., Szczepanski, N., Tabary, S.: An adaptive parallel sat solver. In: International Conference on Principles and Practice of Constraint Programming (CP). pp. 30–48. Springer (2016)
3. Audemard, G., Simon, L.: Refining Restarts Strategies for SAT and UNSAT. In: Principles and Practice of Constraint Programming. pp. 118–126. Springer (2012)
4. Audemard, G., Simon, L.: Glucose and syrup: Nine years in the SAT competitions. Proc. of SAT Competition pp. 24–25 (2018)
5. Balyo, T., Sinz, C.: Parallel satisfiability. In: Handbook of Parallel Constraint Reasoning, pp. 3–29. Springer (2018)
6. Biere, A.: Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2017. Proceedings of SAT Competition pp. 14–15 (2017)
7. Biere, A., Heule, M., van Maaren, H.: Handbook of satisfiability, vol. 185. IOS press (2009)
8. Bouraoui, Z., Cornuéjols, A., Denœux, T., Destercke, S., Dubois, D., Guillaume, R., Marques-Silva, J., Mengin, J., Prade, H., Schockaert, S., et al.: From shallow to deep interactions between knowledge representation, reasoning and machine learning (kay r. amel group). arXiv preprint arXiv:1912.06612 (2019)
9. 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)
10. Guyon, I., Elisseeff, A.: An introduction to variable and feature selection. Journal of machine learning research **3**(Mar), 1157–1182 (2003)
11. Hamadi, Y., Jabbour, S., Sais, L.: Manysat: a parallel SAT solver. Journal on Satisfiability, Boolean Modeling and Computation **6**, 245–262 (2008)
12. Heule, M., Järvisalo, M., Suda, M.: SAT race benchmarks. http://satcompetition.org/sr2019benchmarks.zip (2016)
13. Heule, M., Järvisalo, M., Suda, M.: SAT competition benchmarks. http://sat2018.forsyte.tuwien.ac.at/benchmarks/ (2018)
14. Heule, M., Järvisalo, M., Tomáš, B.: SAT competition benchmarks. http://baldur.iti.kit.edu/sat-competition-2016/index.php?cat=downloads (2016)
15. Heule, M., Järvisalo, M., Tomáš, B.: SAT competition benchmarks. https://baldur.iti.kit.edu/sat-competition-2017/index.php?cat=benchmarks (2017)
16. Heule, M.J., Kullmann, O., Biere, A.: Cube-and-conquer for satisfiability. In: Handbook of Parallel Constraint Reasoning, pp. 31–59. Springer (2018)
17. Heule, M.J., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Haifa Verification Conference. pp. 50–65. Springer (2011)
18. Khalil, E., Dai, H., Zhang, Y., Dilkina, B., Song, L.: Learning combinatorial optimization algorithms over graphs. In: Advances in Neural Information Processing Systems. pp. 6348–6358 (2017)
19. Kurin, V., Godil, S., Whiteson, S., Catanzaro, B.: Improving sat solver heuristics with graph networks and reinforcement learning. arXiv preprint arXiv:1909.11830 (2019)

20. Le Frioux, L., Baarir, S., Sopena, J., Kordon, F.: Painless: a framework for parallel sat solving. In: International Conference on Theory and Applications of Satisfiability Testing (SAT). pp. 233–250. Springer (2017)
21. Le Frioux, L., Baarir, S., Sopena, J., Kordon, F.: Modular and efficient divide-and-conquer sat solver on top of the painless framework. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 135–151. Springer (2019)
22. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for sat solvers. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 123–140. Springer International Publishing (2016)
23. Liang, J.H., Oh, C., Ganesh, V., Czarnecki, K., Poupart, P.: Maple-comsps LRB VSIDS and maplecomsps CHB VSIDS. Proc. of SAT Competition pp. 20–21 (2017)
24. Liang, J.H., Oh, C., Mathew, M., Thomas, C., Li, C., Ganesh, V.: Machine learning-based restart policy for cdcl sat solvers. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 94–110. Springer (2018)
25. Liang, J.H., VK, H.G., Poupart, P., Czarnecki, K., Ganesh, V.: An empirical study of branching heuristics through the lens of global learning rate. In: International Conference on Theory and Applications of Satisfiability Testing (SAT). pp. 119–135. Springer (2017)
26. Liaw, A., Wiener, M., et al.: Classification and regression by randomforest. R news **2**(3), 18–22 (2002)
27. Lin, H.T., Li, L.: Reduction from cost-sensitive ordinal ranking to weighted binary classification. Neural Computation **24**(5), 1329–1367 (2012)
28. Liu, T.Y., et al.: Learning to rank for information retrieval. Foundations and Trends® in Information Retrieval **3**(3), 225–331 (2009)
29. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A Search Algorithm for Propositional Satisfiability. Computers, IEEE Transactions on **48**(5), 506–521 (1999)
30. Mironov, I., Zhang, L.: Applications of sat solvers to cryptanalysis of hash functions. In: International Conference on Theory and Applications of Satisfiability Testing (SAT). pp. 102–115. Springer (2006)
31. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of the 38th annual Design Automation Conference. pp. 530–535. ACM (2001)
32. Nejati, S., Ganesh, V.: Cdcl (crypto) sat solvers for cryptanalysis. arXiv preprint arXiv:2005.13415 (2020)
33. Nejati, S., Liang, J.H., Ganesh, V., Gebotys, C., Czarnecki, K.: SHA-1 preimage instances for SAT. Proc. of SAT Competition p. 45 (2017)
34. Nejati, S., Liang, J.H., Gebotys, C., Czarnecki, K., Ganesh, V.: Adaptive restart and CEGAR-based solver for inverting cryptographic hash functions. In: Working Conference on Verified Software: Theories, Tools, and Experiments. pp. 120–131. Springer (2017)
35. Nejati, S., Newsham, Z., Scott, J., Liang, J.H., Gebotys, C., Poupart, P., Ganesh, V.: A propagation rate based splitting heuristic for divide-and-conquer solvers. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 251–260. Springer (2017)
36. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al.: Scikit-learn: Machine learning in python. Journal of machine learning research **12**(Oct), 2825–2830 (2011)
37. Rintanen, J.: Planning and SAT. Handbook of Satisfiability **185**, 483–504 (2009)
38. Shah, N.B., Wainwright, M.J.: Simple, robust and optimal ranking from pairwise comparisons. The Journal of Machine Learning Research **18**(1), 7246–7283 (2017)

39. Soos, M., Kulkarni, R., Meel, K.S.: CrystalBall: Gazing in the black box of SAT solving. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 371–387. Springer (2019)
40. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: Evaluating component solver contributions to portfolio-based algorithm selectors. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 228–241. Springer (2012)
41. Zhang, H., Bonacina, M.P., Hsiang, J.: Psato: a distributed propositional prover and its application to quasigroup problems. Journal of Symbolic Computation **21**(4-6), 543–560 (1996)