SATzilla-07: The Design and Analysis of an Algorithm Portfolio for SAT

Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown

University of British Columbia, 2366 Main Mall, Vancouver BC, V6T 1Z4, Canada
{xulin730,hutter,hoos,kevinlb}@cs.ubc.ca

Abstract. It has been widely observed that there is no "dominant" SAT solver; instead, different solvers perform best on different instances. Rather than following the traditional approach of choosing the best solver for a given class of instances, we advocate making this decision online on a per-instance basis. Building on previous work, we describe a per-instance solver portfolio for SAT, SATzilla-07, which uses so-called empirical hardness models to choose among its constituent solvers. We leverage new model-building techniques such as censored sampling and hierarchical hardness models, and demonstrate the effectiveness of our techniques by building a portfolio of state-of-the-art SAT solvers and evaluating it on several widely-studied SAT data sets. Overall, we show that our portfolio significantly outperforms its constituent algorithms on every data set. Our approach has also proven itself to be effective in practice: in the 2007 SAT competition, SATzilla-07 won three gold medals, one silver, and one bronze; it is available online at http://www.cs.ubc.ca/labs/beta/Projects/SATzilla.

1 Introduction

The propositional satisfiability problem (SAT) is one of the most fundamental problems in computer science. SAT is interesting for its own sake, but also because instances of other $\mathcal{NP}$-complete problems can be encoded into SAT and solved by SAT solvers. This approach has proven effective for planning, scheduling, graph coloring and software/hardware verification problems. The conceptual simplicity of SAT facilitates algorithm development, and significant research and engineering efforts have led to sophisticated algorithms with highly-optimized implementations. By now, many such high-performance SAT solvers exist. Although there are some general patterns describing which solvers tend to be good at solving certain kinds of instances, it is still often the case that one solver is better than others at solving some problem instances from a given class, but dramatically worse on other instances. Indeed, we know that no solver can be guaranteed to dominate all others on unrestricted SAT instances [2]. Thus, practitioners with hard SAT problems to solve face a potentially difficult algorithm selection problem [27]: which algorithm(s) should be run in order to minimize some performance objective, such as expected runtime?

The most widely-adopted solution to such algorithm selection problems is to measure every candidate solver's runtime on a representative set of problem instances, and then to use only the algorithm which offered the best (e.g., average

or median) performance. We call this the "winner-take-all" approach. Its use has resulted in the neglect of many algorithms that are not competitive on average but that nevertheless offer very good performance on particular instances. The *ideal* solution to the algorithm selection problem, on the other hand, would be to consult an oracle that tells us the amount of time that each algorithm would take to solve a given problem instance, and then to select the algorithm with the best performance.

Unfortunately, computationally cheap, perfect oracles of this nature are not available for SAT or any other $\mathcal{NP}$-complete problem, and we cannot precisely determine an arbitrary algorithm's runtime on an arbitrary instance without actually running it. Nevertheless, our approach to algorithm selection in this paper is based on the idea of building approximate runtime predictors, which can be seen as heuristic approximations to perfect oracles. Specifically, we use machine learning techniques to build an *empirical hardness model*, a computationally inexpensive way of predicting an algorithm's runtime on a given problem instance based on features of the instance and the algorithm's past performance [25, 19]. This approach has previously yielded effective algorithm portfolios for the winner determination problem (WDP) in combinatorial auctions [18, 17]; however, there exist relatively few state-of-the-art solvers for WDP.

To show that algorithm portfolios based on empirical hardness models can also effectively combine larger sets of highly-optimized algorithms, we consider the satisfiability problem in this work. Specifically, we describe and analyze `SATzilla`, a portfolio-based SAT solver that utilizes empirical hardness models for per-instance algorithm selection. `SATzilla` goes back to 2003, when its original version was first submitted to the SAT competition [24, 25]. In that competition, `SATzilla` placed 2$^{\text{nd}}$ in the random instances category, 2$^{\text{nd}}$ in the handmade instances (satisfiable only) category, and 3$^{\text{rd}}$ in the handmade instances category. Here, we describe a substantially improved version of `SATzilla`, which uses new techniques, such as censored sampling and hierarchical hardness models, as well as an updated set of solvers. This new solver, dubbed `SATzilla-07`, was entered into the 2007 SAT competition and placed 1$^{\text{st}}$ in the handmade, handmade (unsatisfiable only) and random categories, 2$^{\text{nd}}$ in the handmade (satisfiable only) category, and 3$^{\text{rd}}$ in the random (unsatisfiable only) category. Here, we give a detailed description and performance analysis for `SATzilla-07`, something which was never published for the original of `SATzilla`.

There exists a fair amount of work related to ours. Lobjois et al. studied the problem of selecting between branch-and-bound algorithms [20] based on an estimate of search tree size due to Knuth. Gebruers et al. employed case-based reasoning to select a solution strategy for instances of a CP problem [8]. One problem with such classification approaches [12] is that they use a misleading error metric, penalizing misclassifications equally regardless of their cost. For the algorithm selection problem, however, using a sub-optimal algorithm is acceptable if the difference between its runtime and that of the best algorithm is small. (Our `SATzilla` approach can be considered to be a classifier with an error metric that depends on the difference in runtime between algorithms.)

Further related work includes "online" approaches that switch between algorithms during runtime. Gomes et al. built a portfolio of stochastic algorithms

for quasi-group completion and logistics scheduling problems [10]; rather than choosing a single algorithm, their approach achieved performance improvements by running multiple algorithms. Lagoudakis & Littman employed reinforcement learning to solve an algorithm selection problem at each decision point of a DPLL solver for SAT in order to select a branching rule [16]. Low-knowledge algorithm control by Carchrae & Beck employed a portfolio of anytime algorithms, prioritizing each algorithm according to its performance so far [3]. Gagliolo & Schmidhuber learned dynamic algorithm portfolios that also support running several algorithms at once [7], where an algorithm's priority depends on its predicted runtime conditioned on the fact that it has not yet found a solution.

2 Building Portfolios with Empirical Hardness Models

The general methodology for building an algorithm portfolio we use in this work follows that of Leyton-Brown et al. [18] in its broad strokes, but we have made significant extensions here. Portfolio construction happens offline, as part of algorithm development, and comprises the following steps:

1. Identify a target distribution of problem instances.
2. Select a set of candidate solvers that have relatively uncorrelated runtimes on this distribution.
3. Use domain knowledge to identify features that characterize problem instances.
4. On a training set of problem instances, compute these features and run each algorithm to determine running times.
5. Optionally, identify one or more solvers to use for pre-solving instances, by examining the algorithms' runtimes. These pre-solvers will later be run for a short amount of time before features are computed (step 9 below), in order to ensure good performance on very easy instances.
6. Using a validation data set, determine which solver achieves the best average runtime (i.e., is the winner-take-all choice) on instances that would not have been solved by the pre-solvers.
7. Construct an empirical hardness model for each algorithm.
8. Choose the best subset of solvers to use in the final portfolio. We formalise and automatically solve this as a simple subset selection problem: from all given solvers, select a subset for which the respective portfolio (which uses the empirical hardness models learned in the previous step) achieves the lowest total runtime on the validation set.

Then, online, to solve a given instance, the following steps are performed:

9. Optionally, run each pre-solver for up to some fixed cutoff time.
10. Compute feature values. If feature computation cannot be finished for some reason (error, timeout), select the solver identified in step 6 above.
11. Otherwise, predict each algorithm's runtime using the empirical hardness models from step 7 above.
12. Run the algorithm predicted to be fastest. If one solver fails to finish its run (e.g., it crashes), run the algorithm predicted to be next-fastest.

In this work, we apply this general strategy to SAT and consider two different settings. In the first, discussed in Section 4, we investigate a problem distribution based on SAT-encoded quasi-group completion instances, which we obtained from an existing generator. On this fairly homogeneous distribution, we attempt to minimize average runtime. In our second setting, discussed in Section 5, we study several different distributions defined by sets of representative instances: the different categories of the SAT competition. These distributions are all highly heterogeneous. Our goal here is to maximize the SAT competition's scoring function: $Score(P, S_i) = 1000 \cdot SF(P, S_i)/\Sigma_j SF(P, S_j)$, where the speed factor $SF(P, S) = timeLimit(P)/(1 + timeUsed(P, S))$ reflects the fraction of the maximum time allowed for an instance S that was used by solver P.[1] Notice that when minimizing average runtime it does not much matter which solver is chosen for an easy instance on which all solvers are relatively fast, as the overall average will remain essentially unchanged. Given the competition's scoring function, however, we must *always* strive to choose the fastest algorithm.

3 Constructing Empirical Hardness Models

The success of an algorithm portfolio built using the methodology above depends on our ability to learn empirical hardness models that can accurately predict a solver's runtime for a given instance using efficiently computable features. In experiments presented in this paper, we use the same ridge regression method (linear in a set of quadratic basis functions) that has previously proven to be very successful in predicting runtime on uniform random k-SAT and on combinatorial auction winner determination [25, 19]. Other learning techniques (e.g., lasso regression, SVM regression, and Gaussian process regression) are also possible; it should be noted that our portfolio methodology is independent of the method used for estimating an algorithm's runtime.

Feature selection and ridge regression. To predict the runtime of an algorithm $\mathcal{A}$ on an instance distribution $\mathcal{D}$, we run algorithm $\mathcal{A}$ on n instances drawn from $\mathcal{D}$ and compute for each instance i a set of features $\boldsymbol{x}_i = [x_{i,1}, \ldots, x_{i,m}]$. We then fit a function $f(\boldsymbol{x})$ that, given the features $\boldsymbol{x}_i$ of instance i, yields a prediction, y_i, of $\mathcal{A}$'s runtime on i. Unfortunately, the performance of learning algorithms can suffer when some features are uninformative or highly correlated with other features, and in practice both of these problems tend to arise. Therefore, we first reduce the set of features by performing feature selection, in our case forward selection. Next, we perform a quadratic basis function expansion of our feature set to obtain additional pairwise product features $x_{i,j} \cdot x_{i,k}$ for $j = 1 \ldots m$ and $k = j+1 \ldots m$. Finally, we perform another pass of forward selection on this extended set to determine our final set of basis functions, such that for instance i we obtain an expanded feature vector $\boldsymbol{\phi}_i = \boldsymbol{\phi}(\boldsymbol{x}_i) = [\phi_1(\boldsymbol{x}_i), \ldots, \phi_d(\boldsymbol{x}_i)]$, where d is the number of basis functions. We then use ridge regression to fit the free parameters $\boldsymbol{w}$ of the function $f_{\boldsymbol{w}}(\boldsymbol{x})$ as follows. Let $\tilde{\boldsymbol{y}}$ be a vector with $\tilde{y}_i = \log y_i$. Let $\boldsymbol{\Phi}$ be an $n \times d$ matrix containing the vectors $\boldsymbol{\phi}_i$ for each instance in the

[1] Please see http://www.satcompetition.org/2007/rules07.html for details.

training set, and let I be the identity matrix. Finally, let δ be a (small) regularization constant (to penalize large coefficients $\boldsymbol{w}$ and thereby increase numerical stability). Then, we compute $\boldsymbol{w} = (\delta I + \boldsymbol{\Phi}^\top \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^\top \tilde{\boldsymbol{y}}$. Given a previously unseen instance j, a log runtime prediction is obtained by computing the instance features $\boldsymbol{x}_j$ and evaluating $f_{\boldsymbol{w}}(\boldsymbol{x}_j) = \boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}_j)$.

Accounting for censored data. As is common with heuristic algorithms for solving $\mathcal{NP}$-complete problems, SAT algorithms tend to solve some instances very quickly, while taking an extremely long amount of time to solve other instances. Indeed, this property of SAT solvers is precisely our motivation for building an algorithm portfolio. However, this property has a downside: runtime data can be very costly to gather, as individual runs can literally take weeks to complete, even when other runs on instances of the same size take only milliseconds. The common solution to this problem is to "censor" some runs by terminating them after a fixed cutoff time.

The bias introduced by this censorship can be dealt with in three ways. (We evaluate these three techniques experimentally in Section 4; here we discuss them conceptually.) First, censored data points can be discarded. Since the role of empirical hardness models in an algorithm portfolio can be seen as warning us away from instance-solver pairs that will be especially costly, this approach is highly unsatisfactory—the hardness models cannot warn us about parts of the instance space that they have never seen.

Second, we can pretend that all censored data points were solved at exactly the cutoff time. This approach is better, as it does record hard cases and recognizes them as being hard. (We have used this approach in past work.) However, it still introduces bias into hardness models by systematically underestimating the hardness of censored instances.

The third approach is to build models that do not disregard censored data points, but do not pretend that the respective runs terminated successfully at the cutoff time either. This approach has been extensively studied in the "survival analysis" literature in statistics, which originated in actuarial questions such as estimating a person's lifespan given mortality data and the ages and features of other people still alive. (Observe that this problem is the same as ours, except that for us data points are always censored at the same value. This subtlety turns out not to matter.) Gagliolo et al. showed that censored sampling can have substantial impact on the performance of restart strategies for solving SAT [21]. Different from their solution, we chose the simple, yet effective method by Schmee & Hahn [28] to deal with censored samples. In brief, this method consists of repeating the following steps until convergence:

1. Estimate the runtime of censored instances using the hardness model, conditioning on the fact that each runtime equals or exceeds the cutoff time.
2. Train a new hardness model using true runtimes for the uncensored instances and the predictions generated in the previous step for the censored instances.

Using hierarchical hardness models. This section summarizes ideas from a companion paper [30]. Previous research on empirical hardness models for SAT has shown that we can achieve better prediction accuracy and simpler models than with models trained on mixed instance sets ("unconditional models";

M_{uncond}) if we restrict ourselves to only satisfiable or unsatisfiable instances [25]. Of course, in practice we cannot divide instances in this way; otherwise we would not need to run a SAT solver in the first place. The idea of a hierarchical hardness model is to first predict an instance's satisfiability using a classification algorithm, and then to predict its hardness conditioned on the classifier's prediction. We use the Sparse Multinomial Logistic Regression (SMLR) classifier [14], but any classification algorithm that returns the probability of belonging to each class could be used. We train empirical hardness models (M_{sat}, M_{unsat}) using quadratic basis-function regression for both satisfiable and unsatisfiable training instances. Then we train a classifier to predict the probability that an instance is satisfiable. Finally, we build hierarchical hardness models using a mixture-of-experts approach with clamped experts: M_{sat} and M_{unsat}. We evaluate both models on test data, and weight each model's prediction by the predicted usefulness of that model. (Further details can be found in the companion paper [30].)

4 Evaluating `SATzilla-07` on the `QCP` Data Set

In the following, we will describe the design and empirical analysis of an algorithm portfolio for solving relatively homogeneous QCP instances. The primary motivation for this part of our work was to demonstrate how our approach works in a relatively simple, yet meaningful, application scenario. At the same time, we did not want to make certain aspects of this application, such as solver selection, overly specific to the QCP instance distribution. The equally important goal of a full-scale performance assessment is addressed in Section 5, where we apply `SATzilla-07` to a broad range of instances from past SAT competitions.

1. Selecting instances. In the quasi-group completion problem (QCP), the objective is to determine whether the unspecified entries of a partial Latin square can be filled to obtain a complete Latin square. QCP instances are widely used in the SAT community to evaluate the performance of SAT solvers. We generated 23 000 QCP instances around the solubility phase transition, using the parameters given by Gomes & Selman [9]. Specifically, the order n was drawn uniformly from the interval $[26, 43]$, and the number of holes H (open entries in the Latin square) was drawn uniformly from $[1.75, 2.3] \times n^{1.55}$. We then converted the respective QCP instances to SAT CNF format. On average, the SAT instances in the resulting `QCP` data set have 3 784 variables and 37 755 clauses, but there is significant variability across the set. As expected, there were almost equal numbers of satisfiable and unsatisfiable instances (50.3% *vs* 49.7%).

2. Selecting candidate solvers. In order to build a strong algorithm portfolio, it is necessary to choose solvers whose runtimes are relatively uncorrelated. We have tended to find that solvers designed for different problem domains are less correlated than solvers designed for the same domain. On `QCP`, there is very little runtime correlation (Pearson's $r = -0.055$) between `Eureka` [23] (`INDUSTRIAL`) and `OKsolver` [15] (`RANDOM`), which makes these two solvers perfect candidates for `SATzilla-07`. On the other hand, the runtime correlation between `Eureka` (`INDUSTRIAL`) and `Zchaff_Rand` [22] (`INDUSTRIAL`) is much higher ($r = 0.81$), though still low enough to be useful.

These solvers were chosen because they are known to perform well on various types of SAT instances (as can be seen, e.g., from past SAT competition results). It should be noted, however, that on QCP, they are dominated by other solvers, such as Satzoo [5]; nevertheless, as previously explained, our goal in this evaluation was not to construct a highly QCP-specific portfolio, but to demonstrate and validate our general approach.

3. Choosing features. Instance features are very important for building accurate hardness models. Good features should correlate well with (solver-specific) instance hardness, and they should be cheap to compute, since feature computation time counts as part of SATzilla-07's runtime.

Nudelman et al. [25] described 84 features for SAT instances. These features can be classified into nine categories: problem size, variable-clause graph, variable graph, clause graph, balance features, proximity to Horn formulae, LP-based, DPLL probing, and local search probing. For the QCP data set, we ignored all LP-based features, because they were too expensive to compute. After eliminating features that were constant across our instance set, we ended up with 70 raw features. The computation time for the local search and DPLL probing features was limited to 4 CPU seconds each.

4. Computing features and runtimes. All our experiments were performed using a computer cluster consisting of 55 machines with dual Intel Xeon 3.2GHz CPUs, 2MB cache and 2GB RAM, running Suse Linux 9.1. All runs of any solver that exceeded 1 CPU hour were aborted (censored). The time for computing all features of a given instance was 13 CPU seconds on average and never exceeded 60 CPU seconds.

We randomly split our data set into training, validation and testing sets at a ratio of 70:15:15. All parameter tuning was performed on the validation set, and the test set was used only to generate the final results reported here. Although test and validation sets of 15% might seem small, we note that each of them contained 3 450 instances.

5. Identifying pre-solvers. Since in this experiment, our goal was simply to minimize expected runtime, a pre-solving step was unnecessary.

6. Identifying the winner-take-all algorithm. We computed average runtimes for all solvers on the training data set, using the cutoff time of 3600 CPU seconds for unsuccesful runs and discarding those instances that were not solved by any of the solvers (the latter applies to 5.2% of the instances from the QCP instance set). The average runtimes were 546 CPU seconds for OKsolver, 566 CPU seconds for Eureka, and 613 CPU seconds for Zchaff_Rand; thus, OKsolver was identified as the winner-take-all algorithm.

7. Learning empirical hardness models. We learned empirical hardness models as described in Section 3. For each solver, we used forward selection to eliminate problematic features and kept the model with the smallest validation error. This lead to empirical hardness models with 30, 30 and 27 features for Eureka, OKsolver and Zchaff_Rand, respectively. When evaluating these models, we specifically investigated the effectiveness of our techniques for censored sampling and hierarchical models.

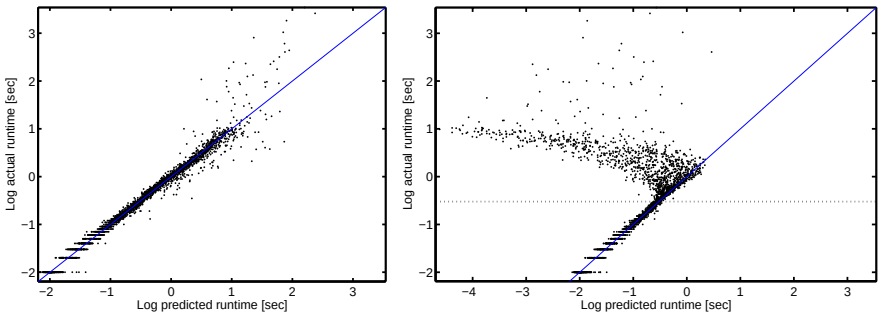

Fig. 1. *Actual vs predicted runtime for* `OKsolver` *on selected (easy) instances from* `QCP` *with cutoff time* $10^{-0.5}$. *Left: trained with complete data; right: censored data points are discarded, RMSE for censored data: 1.713.*

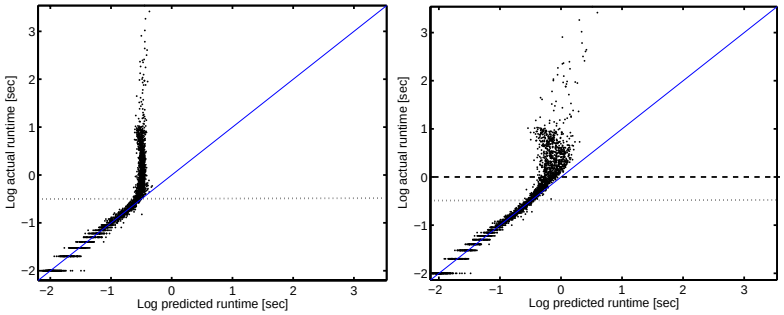

Fig. 2. *Actual vs predicted runtime for* `OKsolver` *on selected (easy) instances from* `QCP` *with cutoff time* $10^{-0.5}$. *Left: set runtime for censored data points as cutoff time, RMSE for censored data: 0.883; right: using the method of Schmee & Hahn [28], RMSE for censored data: 0.608.*

Censored Sampling. We gathered `OKsolver` runtimes on a set of all-satisfiable `QCP` instances of small order. The instances were chosen such that we could determine true runtimes in all cases; we then artificially censored our runtime data, using a cutoff time of $10^{-0.5}$ CPU seconds, and compared the various methods for dealing with censored data surveyed in Section 3 on the resulting data. Fig. 1 (left) shows the runtime predictions achieved by the hardness model trained on ideal, uncensored data (RMSE=0.146). In contrast, Fig. 1 (right) shows that throwing away censored data points leads to very noisy runtime prediction for test instances whose true runtimes are higher than the cutoff time (RMSE for censored data: 1.713). Fig. 2 (left) shows the performance of a model trained on runtime data in which all censored points were labelled as having completed at exactly the cutoff time (RMSE for censored data: 0.883). Finally, Fig. 2 (right) shows that a hardness model trained using the method of Schmee & Hahn [28] yields the best prediction accuracy (RMSE for censored data: 0.608). Furthermore, we see good runtime predictions even for instances where the solver's runtime is up to half an order of magnitude (a factor of three) greater than the cutoff time. When runtimes get much bigger than this, the

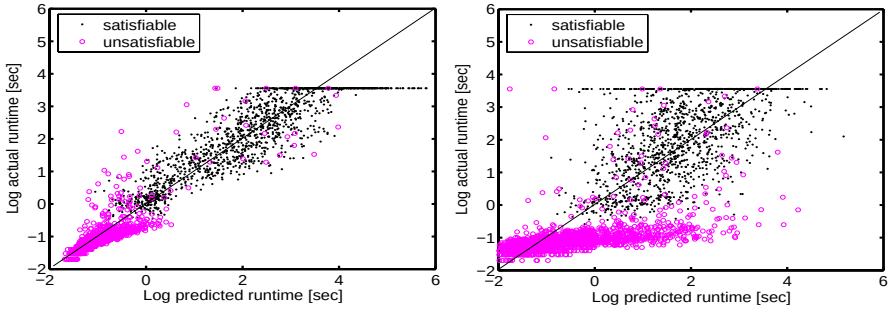

Fig. 3. *Actual vs predicted runtime plots for* `Eureka` *on* `QCP`*. Left: model using a model selection oracle, RMSE=0.630; right: unconditional model, RMSE=1.111.*

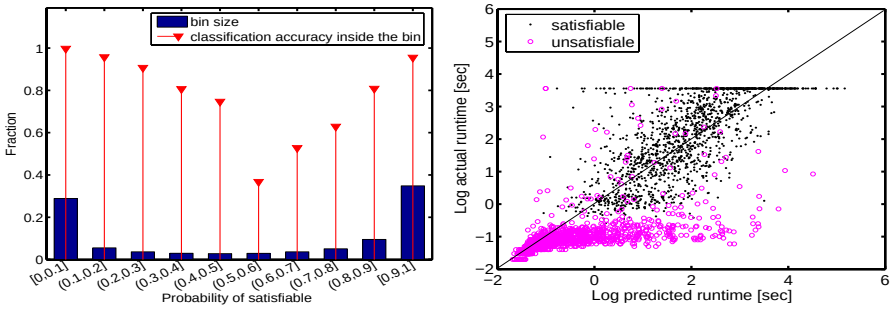

Fig. 4. *Left: Performance of the SMLR classifier on* `QCP`*. Right: Actual vs predicted runtime for* `Eureka` *on* `QCP` *using a hierarchical hardness model, RMSE=0.938.*

prediction becomes much noisier, albeit still better than we observed earlier. (We obtained similar results for the two other solvers, `Eureka` and `Zchaff_Rand`.

Hierarchical Hardness Models. Fig. 3 compares the runtime predictions made by a model with access to a model selection oracle ($M_{oracular}$), and an unconditional model (M_{uncond}) for the `Eureka` solver on the `QCP` instance set. $M_{oracular}$ defines an upper bound on performance with the conditional model. Overall, M_{uncond} tends to make considerably less accurate predictions (RMSE= 1.111) than $M_{oracular}$ (RMSE=0.630). We report the performance of the classifier and hierarchical hardness models in Fig. 4. The overall classification accuracy is 89%; as shown in Fig. 4 (left), the classifier is nearly certain, and usually correct, about the satisfiability of most instances. Although our hierarchical hardness model did not achieve the same runtime prediction accuracy as $M_{oracular}$, its performance is 36% closer to this ideal than M_{uncond} in terms of RMSE. (Note that hierarchical models are not guaranteed to achieve better performance than unconditional models, since the use of the wrong conditional model on certain instances can cause large prediction errors.) Similar results are obtained for the two other solvers, `OKsolver` and `Zchaff_Rand`.

8. Solver subset selection. Using our automated subset selection procedure, we determined that all three solvers performed strongly enough on `QCP` that dropping any of them would lead to reduced portfolio performance.

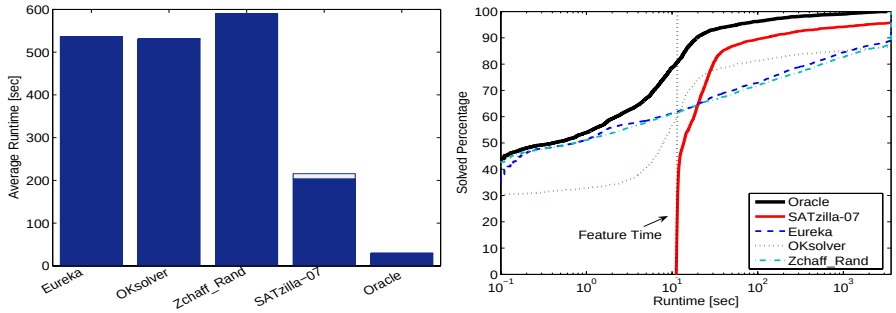

Fig. 5. *Left: Average runtime for different solvers on* `QCP`*; the hollow box for* `SATzilla-07` *represents the time used for computing instance features (13 CPU sec on average). Right: Empirical cumulative distribution functions (CDFs) for the same runtime data.*

9. Performance analysis. We compare the average runtime of `SATzilla-07`, an algorithm selection scheme based on a perfect oracle and all of `SATzilla-07`'s component solvers in Fig. 5 (left). On the test data set, all component solvers have runtimes of around 600 CPU seconds on average; `OKsolver`, the "winner-take-all" choice, has an average runtime of 543 CPU seconds. `SATzilla-07`'s average runtime, 205 CPU seconds, is much lower. Although `SATzilla-07`'s performance is much better than any of its components, it still does significantly worse than the oracle. In particular, it chooses the same solver as the oracle only for 62% of the instances. However, in most of these cases, the runtimes of the solvers picked by `SATzilla-07` and the oracle are very similar, and only for 12% of the `QCP` instances, the solver chosen by `SATzilla-07` is more than 10 CPU seconds slower.

A more nuanced view of the algorithms' empirical performance is afforded by the cumulative distribution functions (CDFs) of their runtimes over the given instance set; these show the fraction of instances that would have been solved if runtime was capped at a given bound. As seen from Fig. 5 (right), for very short runtimes, `SATzilla-07` performs worse than its component solvers, because it requires about 13 CPU seconds (on average) to compute instance features. For higher runtimes, `SATzilla-07` dominates all of its component solvers, and within the 1 CPU hour cutoff, `SATzilla-07` solves about 6% more instances.

5 `SATzilla-07` for the 2007 SAT Competition

In this section, we describe the `SATzilla-07` solvers entered into the 2007 SAT competition and demonstrate that these achieve state-of-the-art performance on a variety of real-world instance collections from past SAT competitions. The purpose of the SAT competitions is to track the state of the art in SAT solving, to assess and promote new solvers, and to identify new challenging benchmarks. In 2007, more than 30 solvers entered the SAT competition. Solvers were scored taking into account both speed and robustness. There were three main categories of instances, `RANDOM`, `HANDMADE` (or `CRAFTED`), and `INDUSTRIAL`.

We submitted three different versions of SATzilla-07 to the 2007 SAT competition. Two versions specifically targeted the RANDOM and HANDMADE categories.[2] In order to study an even more heterogeneous instance distribution, a third version of SATzilla-07 attempted to perform well in all three categories of the competition; we call this meta-category BIG-MIX.

Following our general procedure for portfolio construction (see Section 2), the three versions of SATzilla-07 were obtained as follows.

1. Selecting instances. In order to train empirical hardness models for any of the above scenarios, we required instances that would be similar to those used in the real competition. For this purpose we used instances from the respective categories in all previous SAT competitions, as well as in the 2006 SAT Race (which only featured industrial instances). Instances that were repeated in previous competitions were also repeated in our data sets. Overall, there are 4 811 instances (all of them used in BIG-MIX), 2 300 instances in category RANDOM and 1 490 in category HANDMADE. About 67.5% of these instances can be solved by at least one solver within 1 200 CPU seconds on our reference machine.

2. Selecting solvers. We considered a wide variety of solvers from previous SAT competitions and the 2006 SAT Race for inclusion in our portfolio. We manually analyzed the results of these competitions, selecting all algorithms that yielded the best performance on some subset of instances. Since our focus was on both satisfiable and unsatisfiable instances, we did not choose any incomplete algorithms (with the exception of SAPS as a pre-solver). In the end we selected seven high-performance solvers as candidates for SATzilla-07: Eureka [23], Zchaff_Rand [22], Kcnfs2006 [4], Minisat2.0 [6], March_dl2004 [11], Vallst [29], and Rsat [26]. Since preprocessing has proven to be an important element for some algorithms in previous SAT competitions, we considered seven additional solvers (labeled "+" in the following) that first run the Hyper preprocessor [1], followed by one of the above algorithms on the preprocessed instance. This doubled the number of our component solvers to 14.

3. Choosing features. In order to limit the additional cost for computing features, we limited the total feature computation time per instance to 60 CPU seconds. Again, we used the features of Nudelman et al. [25], but excluded a number of computationally expensive features, such as clause graph and LP-based features. The computation time for each of the local search and DPLL probing features was limited to 1 CPU second. The number of raw features used in SATzilla-07 is 48.

4. Computing features and runtimes. We collected feature and runtime data using the same environment and process as for the QCP data set, we reduced the cutoff time to 1 200 CPU seconds (The same as SAT competition cutoff time).

5. Identifying pre-solvers. Since the scoring function used in the 2007 SAT competition rewards quick algorithm runs, we cannot afford the feature computation for very easy instances (for which runtimes greater than one second

[2] We built a version of SATzilla-07 for the INDUSTRIAL category after the submission deadline and found its performance to be qualitatively similar to the results we present here: on average, it is twice as fast as the best single solver, Eureka.

Solvers	BIG-MIX		RANDOM		HANDMADE	
	Avg. Time	Solved [%]	Avg. Time	Solved [%]	Avg. Time	Solved [%]
Eureka	598	57	770	40	561	59
Kcnfs2006	658	50	319	81	846	33
March_dl2004	394	73	269	85	311	80
Minisat2.0	459	69	520	62	411	73
Rsat	445	70	522	62	412	72
Vallst	620	54	757	40	440	67
Zchaff_Rand	645	51	802	36	562	58
Eureka+	No preprocessing carried out for industrial instances		767	40	556	59
Kcnfs2006+			315	81	824	35
March_dl2004+			269	85	325	79
Minisat2.0+			522	62	421	72
Rsat+			522	62	419	71
Vallst+			753	41	413	71
Zchaff_Rand+			802	37	550	59

Table 1. *Percentage of instances solved by each algorithm and average runtime over all instances solved by at least one solver. "+" means with preprocessing; preprocessing is not carried out for industrial instances as it would often time out.*

are already too large). Thus, we have to solve easy instances before even computing any features. Good algorithms for pre-solving solve a large proportion of instances quickly; based on an examination of the training runtime data we chose March_dl2004 and the local search algorithm SAPS (UBCSAT implementation with the best fixed parameter configuration identified by Hutter et al. [13]) as pre-solvers. Within 5 CPU seconds on our reference machine, March_dl2004 solved 47.8%, 47.7%, and 43.4% of the instances in our RANDOM, HANDMADE and BIG-MIX data sets, respectively. For the remaining instances, we let SAPS run for 2 CPU seconds, because we found its runtime to be almost completely uncorrelated with March_dl2004 ($r = 0.118$ for the 487 remaining instances solved by both solvers). SAPS solved 28.8%, 5.3%, and 14.5% of the remaining RANDOM, HANDMADE and BIG-MIX instances, respectively.

6. Identifying winner-takes-all algorithm. Each solver's performance is reported in Table 1; as can be seen from this data, the winner-take-all solvers for BIG-MIX, RANDOM and HANDMADE happened always to be March_dl2004.

7. Learning empirical hardness models. We learned empirical hardness models as described in Section 3, using the Schmee & Hahn [28] procedure for dealing with censored data as well as hierarchical empirical hardness models [30].

8. Solver subset selection. Based on the results of automatic exhaustive subset search as outlined in Section 2, we obtained portfolios comprising the following solvers for our three data sets:

- BIG-MIX: Eureka, kcnfs2006, March_dl2004, Rsat;
- RANDOM: March_dl2004, kcnfs2006, Minisat2.0+;
- HANDMADE: March_dl2004, Vallst, March_dl2004+, Minisat2.0+, Zchaff_Rand+.

9. Performance analysis. For all three data sets we obtained excellent results: SATzilla-07 always outperformed all its constituent solvers in terms of average runtime and instances solved at any given time. The SAT/UNSAT classifier was surprisingly effective in predicting satisfiability of RANDOM instances, where it reached a classification accuracy of 94%. For HANDMADE and BIG-MIX, the classification accuracy was still at a respectable 70% and 78% (i.e., substantially better than random guessing).

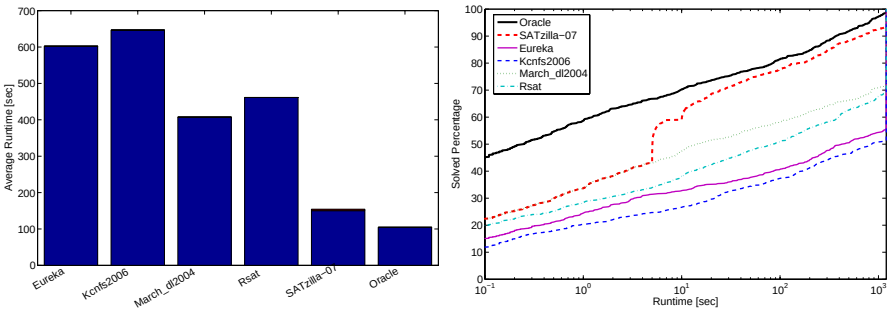

Fig. 6. *Left: Average runtime, right: runtime CDF for different solvers on* `BIG-MIX`*; the average feature computation time was 6 CPU seconds.*

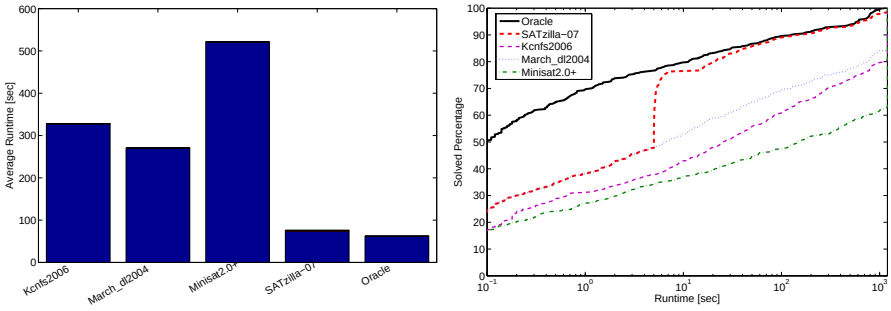

Fig. 7. *Left: Average runtime, right: runtime CDF for different solvers on* `RANDOM`*; the average feature computation time was 3 CPU seconds.*

For `BIG-MIX`, the frequencies with which each solver was selected by a perfect oracle and `SATzilla-07` were found to be similar. However, this does not mean that our hardness models made perfect predictions. Only for 46% of the instances, `SATzilla-07` picked exactly the same solver as the oracle, but it selected a "good solver" (no more than 10 CPU seconds slower) for 89% of the instances. This indicates that many of the mistakes made by our models occur in situations where it does not matter much, because the selected and the best algorithms have very similar runtimes. Although the runtime predictions were not perfect, `SATzilla-07` achieved very good performance (see Fig. 6). Its average runtime (154 CPU seconds) was half that of the best single solver, `March_dl2004` (407 CPU seconds), and it solved 20% more instances than any single solver within the given time limit.

For data set `RANDOM`, Fig. 7 (left) shows that `SATzilla-07` performed much better than the best solver, `March_dl2004`. `SATzilla-07` was more than three times faster on average than the best single solver. The runtime CDF plot (Fig. 7, right) shows that local search pre-solver `Saps` really helps a lot, and `SATzilla-07` dominated `March_dl2004`; in particular, for the same overall cutoff time, `SATzilla-07` solved 15% more instances.

The performance results for `HANDMADE` were very good too. Using five component solvers, `SATzilla-07` was more than 50% faster on average than the

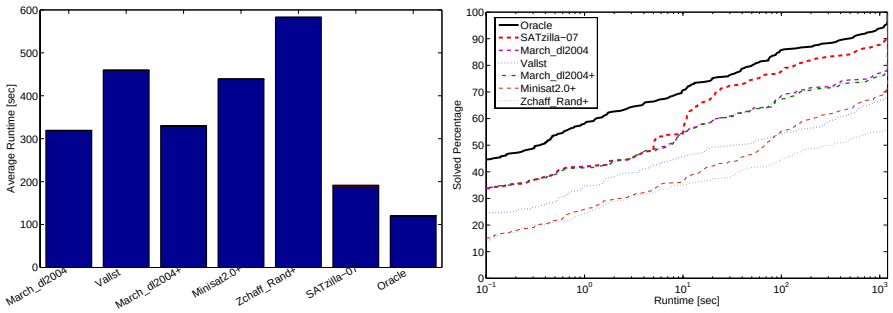

Fig. 8. *Left: Average runtime, right: runtime CDF for different solvers on* `HANDMADE`*; the average feature computation time was 3 CPU seconds.*

best single solver (see Fig. 8, left). The CDF plot in Fig. 8 (right) shows that `SATzilla-07` dominated all its components and solved 11% more instances than the best single solver; overall, its performance was found to be very close to that of the oracle.

6 Conclusions

Algorithms can be combined into portfolios to build a whole greater than the sum of its parts. In this work, we have significantly extended earlier work on algorithm portfolios for SAT that select solvers on a per-instance basis using empirical hardness models for runtime prediction. We have demonstrated the effectiveness of our new portfolio construction method, `SATzilla-07`, on a large set of SAT-encoded QCP instances as well as on three large sets of SAT competition instances. Our own experiments show that our `SATzilla-07` portfolio solvers always outperform their components. Furthermore, `SATzilla-07`'s excellent performance in the recent 2007 SAT competition demonstrates the practical effectiveness of our portfolio approach. `SATzilla` is an open project. We believe that with more solvers and training data added, `SATzilla`'s performance will continue to improve. `SATzilla-07` is available online at http://www.cs.ubc.ca/labs/beta/Projects/SATzilla.

References

1. F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *SAT-03*, pages 341–355, 2003.
2. D. P. Bertsekas. *Linear Network Optimization, Algorithms and Codes*. MIT Press, Cambridge, MA, 1991.
3. T. Carchrae and J. C. Beck. Applying machine learning to low-knowledge control of optimization algorithms. *Computational Intelligence*, 21(4):372–387, 2005.
4. O. Dubois and G. Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *IJCAI-01*, pages 248–253, 2001.
5. N. Eén and N. Sörensson. An extensible SAT solver. In *SAT-03*, pages 502–518, 2003.
6. N. Eén and N. Sörensson. Minisat v2.0 (beta). Solver description, SAT Race, 2006.
7. M. Gagliolo and J. Schmidhuber. Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence*, 47(3-4):295–328, 2007.

8. C. Gebruers, B. Hnich, D. Bridge, and E. Freuder. Using CBR to select solution strategies in constraint programming. In *ICCBR-05*, pages 222–236, 2005.
9. C. P. Gomes and B. Selman. Problem structure in the presence of perturbations. In *AAAI-97*, pages 221–226, 1997.
10. C. P. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.
11. M. Heule and H. V. Maaren. march_dl: Adding adaptive heuristics and a new branching strategy. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:47–59, 2006.
12. E. Horvitz, Y. Ruan, C. P. Gomes, H. Kautz, B. Selman, and D. M. Chickering. A Bayesian approach to tackling hard computational problems. In *Proc. of UAI-01*, pages 235–244, 2001.
13. F. Hutter, Y. Hamadi, H. H. Hoos, and K. Leyton-Brown. Performance prediction and automated tuning of randomized and parametric algorithms. In *Proc. of CP-06*, pages 213–228, 2006.
14. B. Krishnapuram, L. Carin, M. Figueiredo, and A. Hartemink. Sparse multinomial logistic regression: Fast algorithms and generalization bounds. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 957–968, 2005.
15. O. Kullmann. Investigating the behaviour of a SAT solver on random formulas. http://cs-svr1.swan.ac.uk/∼csoliver/Artikel/OKsolverAnalyse.html, 2002.
16. M. G. Lagoudakis and M. L. Littman. Learning to select branching rules in the DPLL procedure for satisfiability. In *LICS/SAT*, volume 9, pages 344–359, 2001.
17. K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. Boosting as a metaphor for algorithm design. In *CP-03*, pages 899–903, 2003.
18. K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. A portfolio approach to algorithm selection. In *IJCAI-03*, pages 1542–1543, 2003.
19. K. Leyton-Brown, E. Nudelman, and Y. Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *CP-02*, pages 556–572, 2002.
20. L. Lobjois and M. Lemaître. Branch and bound algorithm selection by performance prediction. In *AAAI-98*, pages 353 – 358, 1998.
21. J. Schmidhuber M. Gagliolo. Impact of censored sampling on the performance of restart strategies. In *CP-06*, pages 167–181, 2006.
22. Y. S. Mahajan, Z. Fu, and S. Malik. Zchaff2004: an efficient SAT solver. In *SAT-05*, pages 360–375, 2005.
23. A. Nadel, M. Gordon, A. Palti, and Z. Hanna. Eureka-2006 SAT solver. Solver description, SAT Race, 2006.
24. E. Nudelman, A. Devkar, Y. Shoham, K. Leyton-Brown, and H. Hoos. Satzilla: An algorithm portfolio for SAT. Solver description, SAT competition 2004, 2004.
25. E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham. Understanding random SAT: Beyond the clauses-to-variables ratio. In *CP-04*, pages 438–452, 2004.
26. K. Pipatsrisawat and A. Darwiche. Rsat 1.03: SAT solver description. Technical Report D-152, Automated Reasoning Group, UCLA, 2006.
27. J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.
28. J. Schmee and G. J. Hahn. A simple method for regression analysis with censored data. *Technometrics*, 21(4):417–432, 1979.
29. D. Vallstrom. Vallst documentation. http://vallst.satcompetition.org/index.html, 2005.
30. L. Xu, H. H. Hoos, and K. Leyton-Brown. Hierarchical hardness models for SAT. In *CP-07*, 2007.