Understanding the Empirical Hardness of NP-Complete Problems

Kevin Leyton-Brown Holger H. Hoos Frank Hutter Lin Xu
Department of Computer Science
University of British Columbia
201-2366 Main Mall, BC V6T 1Z4, CANADA

Problems are intractable when they "can be solved, but not fast enough for the solution to be usable" [13]. NP-complete problems are commonly said to be intractable; however, the reality is more complex. All known algorithms for solving NP-complete problems require exponential time in the worst case; however, these algorithms nevertheless solve many problems of practical importance astoundingly quickly, and are hence relied upon in a broad range of applications. The propositional satisfiability problem (SAT) serves as a good example. One of the most popular approaches for the formal verification of hardware and software relies on general-purpose SAT solvers and SAT encodings, typically with hundreds of thousands of variables. These instances can often be solved in seconds [34], even though the same solvers can be stymied by hand-crafted instances involving only hundreds of variables.

Clearly, we could benefit from a more nuanced understanding of algorithm behaviour than is offered by asymptotic, worst-case analysis. Our work asks the question most relevant to an end user: *"How hard is it to solve a given family of problem instances, using the best available methods?"* Formal, complexity-theoretic analysis of this question seems hopeless: the best available algorithms are highly complex (and, in some cases, only available in compiled form), and instance distributions representative of practical applications are heterogeneous and richly structured. For this reason, we turn to statistical, rather than combinatorial, analysis.

The main claim of this article is that rigorous statistical methods can characterize algorithm runtime with high levels of confidence. More specifically, this article surveys over a decade of research[1] showing how to build *empirical hardness models (EHMs)* that, given a new problem instance, estimate the runtime of an algorithm in low-order polynomial time [28, 27, 26, 32, 33, 16, 38, 40, 29, 14, 18, 19, 39, 21]. We have shown that it is possible to build quite accurate models for different NP-complete problems (we have studied SAT, combinatorial auction winner determination, mixed integer programming, and the traveling salesman problem), distributions of problem instances (we have considered dozens), solvers (again, dozens).

We have robustly found that even very succinct EHMs can achieve high accuracies, meaning that they describe simple relationships between instance characteristics and algorithm runtime. This makes our approach important even for theoretically inclined computer scientists who prefer proofs to experimental findings: EHMs can uncover new, simple relationships between instance characteristics and runtime, and thereby catalyze new theoretical work.

The focus of this paper is on ways that EHMs contribute to our *understanding* of NP-complete problems; however, they are also useful in a variety of practical applications. Most straightforwardly, they can aid the distribution of problem instances across a cluster, or predict how long a run will take to complete. More interestingly, they can (1) be used to combine a set of high-variance algorithms into an "algorithm portfolio" that outperforms its constituents; (2) be leveraged to automatically make benchmark distributions more challenging; and (3) aid in the configuration (or "tuning") of highly parameterized algorithms for good performance on given instance distributions. More detailed explanations of these applications appear in sidebars throughout this article.

Phase Transitions in Uniform-Random 3-SAT

We begin by describing the most widely known relationship between a characteristic of fixed-size random SAT instances and solver runtime. (After this, we consider more realistic instances of SAT and other NP-hard problems.) Let $p(c, v)$ denote the probability that a satisfiable 3-SAT formula[2] will be generated by uniformly sampling c clauses of 3 variables each from a set of v variables, negating each with probability 0.5. In the early 1990s, researchers discovered that when v is held constant, $p(c, v)$ exhibits a "phase transition" as c/v crosses a critical value of about 4.26 [8, 31]. Intuitively, instances with few clauses are underconstrained and thus almost always satisfiable, while those with many clauses are overconstrained and thus almost always unsatisfiable. The interesting fact is that, for all fixed values of v so far tested, the phase transition point at which $p(c, v)$ is exactly 0.5, appears to coincide with a runtime peak even for the SAT solvers that perform best on these instances. This finding thus links an algorithm-independent property of an instance (c/v) with algorithm-specific runtime in a way that has proven robust across solvers.

Figure 1 (Left) shows this relationship using real data. The dotted line shows $p(c, v)$ for uniform-random 3-SAT instances with $v = 400$, while the solid line shows the mean runtime of `march_hi` [11], one of the best SAT solvers for uniform-random 3-SAT, on the same instances. We do indeed observe both a phase transition and a hardness spike at the phase transition point. However, there is more to the story. Figure 1 (Right) plots raw runtime data (on a log scale) for `march_hi`, with each point corresponding to a single (random) 3-SAT formula. We can now see that the c/v ratio does not suffice to fully explain `march_hi`'s empirical behaviour on these instances: there is still substantial variation at each point along the x axis, with over two orders of magnitude at the "hard" phase transition

[1]Some work described in this article was performed with additional coauthors: Eugene Nudelman and Yoav Shoham made particularly sustained contributions, and Galen Andrew, Alex Devkar, and Jim McFadden also deserve mention. We do not survey the literature on algorithm performance prediction here; instead, we focus on our own work. For extensive discussions of related work, please see [29, 21].

[2]A SAT formula F is solved by deciding whether there exists an assignment of its variables under which F evaluates to true. A subclass of particular importance is 3-SAT. A 3-SAT instance is a conjunction of *clauses*, each of which is a disjunction of 3 variables or their negations. For example, $(v_1 \vee \neg v_2 \vee v_4) \wedge (\neg v_1 \vee \neg v_3 \vee v_4)$ is a simple formula with $v = 4$ variables and $c = 2$ clauses that has several satisfying assignments (*e.g.*, $[v_1, v_2, v_3, v_4] =$ [true, true, false, false]).

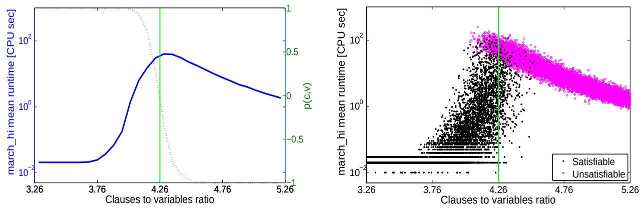

Figure 1: Runtime of `march_hi` on uniform-random 3-SAT instances with $v = 400$ and variable c/v ratio. Left: mean runtime, along with $p(c, v)$; Right: per-instance runtimes, coloured by satisfiability status. Runtimes were measured with an accuracy of 0.01s, leading to the discretization effects visible near the bottom of the figure. Every point represents one SAT instance.

point. The runtime pattern also depends on satisfiability status: hard instances are scarcer and runtime variation is greater among satisfiable instances than among unsatisfiable instances. One reason for this is that on satisfiable instances the solver can stop as soon as it encounters a satisfying assignment, whereas for unsatisfiable instances a solver must prove that no satisfying assignment exists anywhere in the search tree.

A Case Study on Uniform-Random 3-SAT

We now ask whether we can better understand the relationship between instance structure and solver runtime by considering instance features beyond just c/v. We will then use a machine learning technique to infer a relationship between these features and runtime. Formally, we start with a set I of instances, a vector $\mathbf{x_i}$ of feature values for each $i \in I$, and a runtime observation y_i for each $i \in I$, obtained by running a given algorithm on i. Our goal will be to identify a mapping $f : \mathbf{x} \mapsto y$ that predicts y_i as accurately as possible, given $\mathbf{x_i}$. We call such a mapping an empirical hardness model.[3] Observe that we have just described a supervised learning problem, and more specifically a regression problem. There are many different regression algorithms that one could use to solve this problem, and indeed, over the years we have considered about a dozen alternatives. Later in this article we will advocate for a relatively sophisticated learning paradigm (random forests of regression trees), but we begin by discussing a very simple approach: quadratic ridge regression [5]. This method performs linear regression based on the given features and their pairwise products, and penalizes increases in feature coefficients (the "ridge"). We elaborate this method in two ways. First, we transform the response variable by taking its logarithm; this better allows runtimes, which vary by orders of magnitude, to be described by a linear model. Second, we reduce the set of features by performing forward selection: we start with an empty set and iteratively add the feature that (myopically) most improves prediction. The result is simpler, more robust models that are less prone to numerical problems. Overall, we have found that even simple learning algorithms like this one usually suffice to build strong EHMs; more important is identifying a good set of instance features.

Instance features

It can be difficult to identify features that correlate as strongly with instance hardness as c/v. We therefore advocate including all features that show some promise of being predictive, and relying on the machine learning algorithm to identify the most useful ones. Our only requirement is that the features be computable in low-order polynomial time; in some applications, we sometimes restrict ourselves to features that are quadratic time or faster. For the SAT domain, we defined 138 features, summarized as follows:

- **Problem size measures** c and v, plus nonlinear combinations we expected to be important, like c/v and $c/v - 4.26$;
- **Syntactic properties** of the instance (proximity to Horn clauses; balance of positive and negative literals; etc.);
- **Constraint graph statistics.** We considered three graphs: nodes for variables and edges representing shared constraints (clauses); nodes for clauses and edges representing shared variables with opposite polarity; nodes for both clauses and variables, and edges representing the occurrence of a variable in a given clause. For each graph, we computed various statistics based on node degrees, path lengths, clustering, etc.;
- **A measure of the integrality** of the optimal solution to the linear programming relaxation of the given SAT instance—specifically, the distance between this solution and the nearest (feasible or infeasible) integral point;
- **Knuth's estimate of search tree size** [25];
- **Probing features** computed by running bounded-length trajectories of local search and tree search algorithms and extracting statistics from these probes (*e.g.*, number of steps before reaching a local minimum in local search or amount of unit propagation performed in tree search).

[3]It is sometimes useful to build EHMs that predict a probability distribution over runtimes rather than a single runtime; see [21]. For simplicity, here we discuss only the prediction of mean runtime.

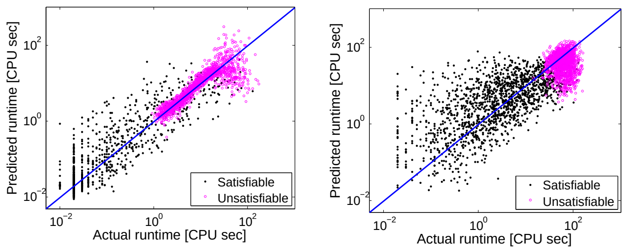

Figure 2: Actual *vs* predicted runtimes for `march_hi` on uniform-random 3-SAT. Each dot represents a test instance not used to train the model; perfect predictions would fall along the diagonal. Left: $c/v \in [3.26, 5.26]$; Right: $c/v = 4.26$.

Model Performance

Let us now investigate the models we can build using these techniques for uniform-random 3-SAT. We consider two sets of instances: one in which the c/v ratio varies around the phase transition point, and another in which it is fixed at $c/v = 4.26$. The first set of instances is less challenging, as we already know that the c/v ratio suffices to explain much of the runtime variation. However, this set is still useful as a sanity check to ensure that our methods do discover the importance of the c/v feature, and to investigate what additional features turn out to be useful. The second set contains fixed-size instances in the hard $c/v = 4.26$ region: any patterns we can find here are interesting since we cannot distinguish instances based on their c/v ratio. In both cases (as with all other empirical results we show in this paper) we randomly partitioned our data into a "training set" that we used to build the EHM and a disjoint "test set" that we used solely to evaluate the performance of the EHM, thereby assessing the accuracy of a model's predictions beyond the data used for constructing it.

Figure 2 shows the results of our investigation, giving true *vs* predicted runtimes of `march_hi`, with every point in the figure corresponding to a different test-set problem instance. Overall, the points cluster around the diagonal in both plots, meaning that the predictions are reasonably accurate. (Indeed, these results are better than they appear to the eye, as a greater density of points falls closer to the diagonal.) Observe that prediction accuracy was considerably better for unsatisfiable instances than for satisfiable instances. Root mean squared error (RMSE) is a numerical measure of a model's accuracy; the two models achieved RMSEs of 0.31 and 0.56, respectively. A model that consistently mispredicted runtimes by a factor of 10 would have achieved an RMSE of 1.0.

We used the variable ratio instance set to verify that the model recognizes the importance of the c/v feature; similarly, we would like to identify other informative features. We cannot do this simply by examining a scatterplot, nor by looking at the coefficients of the model itself: because many of our features are strongly correlated, important features can have small coefficients and unimportant features can have big coefficients. Instead, we identify new models that use only a small number of uncorrelated features by applying the same forward selection method described previously, but terminating much earlier, when the benefit from adding new features begins to taper off. Then (following [10]) we quantify the importance of each feature in these new models by measuring the loss in RMSE incurred by omitting it, and scaling these values so that the most important feature receives a score of 100. Table 1 demonstrates that the model indeed identified c/v (and the variant $|c/v - 4.26|$) as an important feature; recall that our quadratic regression has access to all pairwise products of features. Other features were also important, in particular, SAPS_BestSolution_Mean, the average number of satisfied clauses achieved by short runs of the local search procedure SAPS [20]. This is interesting, as one might not have expected the performance of a local search algorithm to be informative about the performance of a tree search algorithm.

For the fixed-ratio data, c/v was constant, and we therefore see a different picture among important features (Table 2). Again, local search probing features figured prominently (GSAT_BestSolution_Mean gives the average number of satisfied clauses achieved by short runs of GSAT [36]). Another important feature is CG_entropy. It gives the entropy across node degrees in the "constraint graph" in which nodes correspond to clauses and edges indicate that a pair of clauses share one or more variables of opposite sign. We have repeated this analysis for other SAT solvers (e.g., `kcnfs`, `satz`) and obtained the same qualitative results: runtime is predictable with high accuracy, small models suffice for good performance, and local-search and constraint-graph features are important [33].

We have already observed that satisfiable and unsatisfiable instances exhibit very different distributions of algorithm runtimes. We thus considered the problem of building EHMs only for satisfiable or for unsatisfiable instances. (We call these "conditional models" as they depend on knowing the satisfiability of a given instance.) We found that conditional EHMs were more accurate than unconditional EHMs; more interestingly, *single-feature* conditional models turned out to be sufficient for predicting runtime with high accuracy. For satisfiable instances, this feature was GSAT_BestSolution_Mean, whereas for unsatisfiable instances, it was Knuth_Mean. We can explain these findings as follows. Since local search algorithms apply heuristics to find a solution as quickly as possible, the reliability with which such an algorithm is able to make quick progress is informative about the speed at which a complete algorithm will be able to find a solution. Tree search algorithms must rule out every node in the tree to prove unsatisfiability; thus, an estimate of this tree size is the most important feature in the unsatisfiable case.

Feature	Score
$\lvert c/v - 4.26\rvert\times$ SAPS_BestSolution_Mean	100
$c/v\times$ SAPS_BestSolution_Mean	19
GSAT_BestSolution_CoeffVar $\times$ SAPS_BestStep_CoeffVar	19
SAPS_BestStep_CoeffVar $\times$ CG_entropy	18

Table 1: Feature importance, $c/v \in [3.26, 5.26]$.

Feature	Score
GSAT_BestSolution_Mean^2	100
GSAT_BestSolution_Mean	88
SAPS_BestSolution_CoeffVar $\times$ SAPS_AvgImprove_Mean	33
SAPS_BestStep_CoeffVar $\times$ CG_Entropy	22

Table 2: Feature importance, $c/v = 4.26$.

Predicting Satisfiability Status

These observations led us to a new idea: building a classifier that directly predicts satisfiability status, and then leveraging conditional EHMs based on this prediction. There are two reasons to be skeptical about this approach. First, conditional EHMs could make very inaccurate predictions on instances with the "wrong" satisfiability status, so it is not clear that we would obtain improved accuracy overall. Second, and more fundamentally, it may seem doubtful that we could accurately predict satisfiability status—that would correspond to guessing whether an instance is solvable without actually solving it! Despite this reservation, and applying sophisticated statistical techniques to mitigate the potential cost of mispredictions (see [38]), we did build *hierarchical hardness models* for uniform random 3-SAT instances. These models achieved (relatively modest) improvements in predictive accuracy on both the fixed-ratio and variable-ratio instance sets. Clearly, hierarchical hardness models can only outperform regular EHMs if our classifier is able to accurately predict satisfiability status. In the variable-ratio case, a natural baseline is a classifier that predicts satisfiability for instances with $c/v < 4.26$, and unsatisfiability otherwise. This classifier achieved an accuracy of 96%, underlining the predictive power of the c/v feature. Our classifier further increased this accuracy to 98% (halving the error rate). Unlike the baseline, our classifier also applies in the fixed-ratio case, where it achieved accuracy of 86%.

We found this result surprising enough that we investigated it in more depth [39]. We considered instances of varying size, from 100 variables (solvable in milliseconds) to 600 variables (solvable in a day; about the largest instances we could solve practically). We focused on instances generated at the phase transition point, because they pose the hardest prediction problem: the probability of generating satisfiable instances at this point is 50%, and in practice, our data sets were indeed very evenly balanced between satisfiable and unsatisfiable instances. Our aims were to investigate whether prediction accuracy appeared fall to that of random guessing on larger problems, and if not, to give an easily comprehensible model that could serve as a starting point for theoretical analysis. In doing so

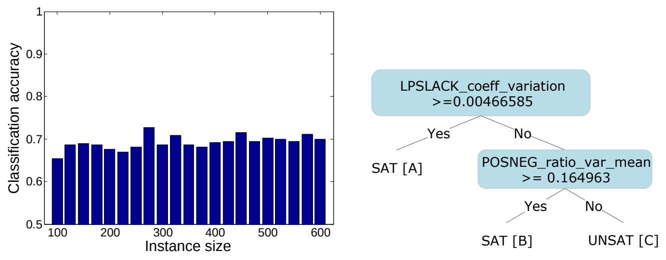

Figure 3: Left: Classification accuracies for our simple decision tree on uniform-random 3-SAT instances at the phase transition with varying numbers of variables. The tree was trained only on 100-variable data. Right: The decision tree. Predictive accuracies for instances falling into the 3 regions were between 60% and 70% (region A); about 50% (region B); and between 70% and 80% (region C).

we restricted our models in three ways that each reduced predictive accuracy, but allowed us to better answer these questions. First, we permitted ourselves to train our classifier only on the 100-variable instances. Second, we considered only decision trees [7] having at most two decision nodes. (We obtained these models by a standard decision tree learning procedure: greedily choosing a feature that best partitioned the training data into satisfiable and unsatisfiable instances, and recursively splitting the resulting partitions.) Finally, we omitted all probing features. Although probing (e.g., local-search) features were very useful for prediction, they were disproportionately effective on small instances and hence would have complicated our study of scaling behaviour. Also, because we aimed to obtain easily comprehensible models, we were disinclined to use features based on complex, heuristic algorithms.

Given all of these restrictions, we were astonished to observe predictive accuracies consistently above 65%, and apparently independent of problem size (see Figure 3 (Left); statistical testing showed no evidence that accuracy falls with problem size). Indeed, our first two restrictions appear to have come at low cost, decreasing accuracies by only about 5%. (Furthermore, after lifting these restrictions, we still found no evidence that accuracy was affected by problem size.) Hence, the reader may be interested in understanding our two-feature model in more detail; we hope that it will serve as a starting point for new theoretical analysis of SAT finite-size instances at the phase transition. The model is given in Figure 3 (Right). LPSLACK_coeff_variation is based on solving a linear programming relaxation of an integer program representation of SAT instances. For each variable i with LP solution value $S_i \in [0, 1]$, $\texttt{LPSLACK}_i$ is defined as $\min\{1 - S_i, S_i\}$: the deviation of S_i from integrality. LPSLACK_coeff_variation is then the coefficient of variation (the standard deviation divided by the mean) of the vector LPSLACK. POSNEG_ratio_var_mean is the average ratio of positive and negative occurrences of each variable. For each variable i with P_i positive occurrences and N_i negative occurrences, $\texttt{POSNEG\_ratio\_var}_i$ is defined as $|0.5 - P_i/(P_i + N_i)|$. POSNEG_ratio_var_mean is then the average over elements of the vector POSNEG_ratio_var. Finally, recall that our model was trained on constant-size instances; we normalized the LPSLACK_coeff_variation and POSNEG_ratio_var_mean features to have mean 0 and standard deviation 1 on this training set. To evaluate the model on a given instance of a different size, we randomly sampled many new instances of that size to compute new normalization factors, which we then applied to the given instance.

Application: Algorithm Selection (SATzilla)

There currently exists no "best" SAT solver; different solvers perform well on different families of instances, and performance differences between them are typically very large. The effectiveness of EHMs suggests a straightforward solution to the *algorithm selection problem* [35]: given a new problem instance, predict the runtime of several SAT solvers, and then run the one predicted to be fastest. This approach [27] forms the core of SATzilla [32, 33, 40], a portfolio-based algorithm selector for SAT.

SATzilla first participated in the 2003 SAT Competition (http://www.satcompetition.org), and placed second and third in several categories. We have since extensively improved the approach, allowing randomized and local search algorithms as component solvers; introducing the idea of pre-solvers that are run for a short, fixed time before the selected solver; adding the ability to optimize for complex scoring functions; and automating the construction of the selector (*e.g.*, pre-solver selection; component solver selection) given data. Leveraging these improvements, and benefiting from the continued improvement of the component solvers upon which it draws, SATzilla led the field in the 2007 and 2009 SAT Competitions, winning 5 medals each time.

More recently, our design of SATzilla evolved from selection based on runtime predictions (EHMs) to a cost-sensitive classification approach that directly selects the best-performing solver without predicting runtime [41]. In the 2012 SAT Challenge (http://baldur.iti.kit.edu/SAT-Challenge-2012), SATzilla was eligible to enter four categories; it placed first in three of these and second in the fourth. Overall, SATzilla's success demonstrates the effectiveness of automated, statistical methods for combining existing solvers—including "uncompetitive" solvers with poor average performance. Except for the instance features used by our models, our approach is entirely general, and is likely to work well for other problems with high runtime variation. All of our software is publicly available; see http://www.cs.ubc.ca/labs/beta/Projects/SATzilla.

Beyond Uniform-Random 3-SAT

Our original motivation involved studying real problems faced by a practitioner, problems that are very unlikely to have uniform random structure. Thus, it is important to demonstrate that EHMs work reliably for a wide range of more realistic instance distributions, and that they are not limited to SAT. In short, they do, and they aren't. By now, we have built EHMs for four different NP-complete problems: SAT [33, 16, 38, 14, 40, 21], the combinatorial auction winner determination problem (WDP) [28, 29], mixed integer programming (MIP, a standard encoding for problems with both discrete and continuous variables) [14, 19, 21], and the traveling salesman problem (TSP) [21]. Observe that we have considered both optimization and decision problems, and that these problems involve discrete variables, continuous variables, and combinations of the two. For each problem, we derived a new set of instance features. This was not trivial, but not terribly difficult either; in all cases we used problem size measures, syntactic properties, and probing features. Extending what we know now to a new domain would probably entail a few days of work. In our publications and other technical work (e.g., submissions to SAT competitions) we have considered more than 30 instance distributions.

These include sophisticated random generators (*e.g.*, SAT reductions from graph coloring and factoring; combinatorial auction benchmarks based on economic models); instance sets from public bench-

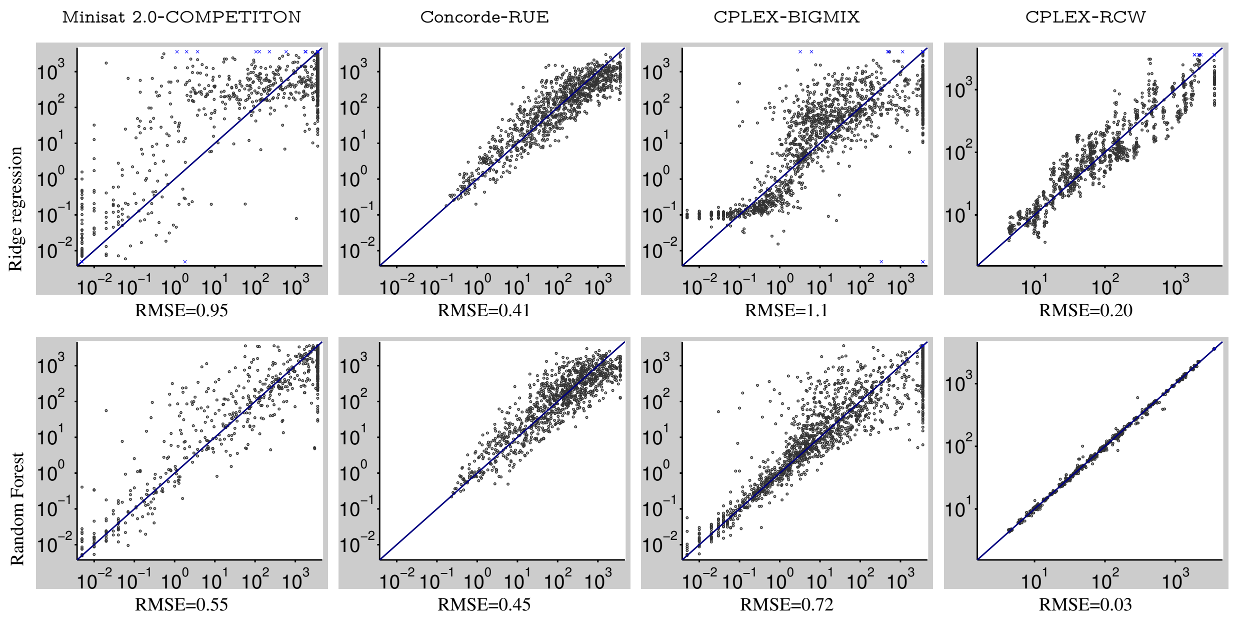

Figure 4: Visual comparison of models for runtime predictions on unseen instances. In each plot, the x-axis denotes true runtime and the y-axis runtime as predicted by the respective model. Predictions above $3\,000$ or below 0.001 are denoted by a blue x.

marks and competitions (*e.g.*, MIPLIB; the SAT competition); and sets of instances derived from practical applications (*e.g.*, SAT-encoded instances from software verification and bounded model checking; industrial MIP instances ranging from machine job allocation to wildlife conservation planning; TSP instances representing drilling circuit boards and traveling between real cities). We have also studied more than 50 state-of-the-art solvers, both open-source projects and proprietary tools developed by industry. Our solvers were both deterministic and randomized, and both complete (*i.e.*, guaranteed to find a solution if one exists) and incomplete. In many cases, we only had access to an executable of the solver, and in no case did we make use of knowledge about a solver's inner workings.

As mentioned earlier, we have also gone beyond quadratic basis function regression to study more than a dozen other statistical modeling techniques, including lasso regression, multivariate adaptive regression splines, support vector machine regression, neural networks, Gaussian processes, regression trees and random forests (see [29, 21]). We omit the details here, but state the conclusion: we now prefer random forests of regression trees [6], particularly when the instance distribution is heterogeneous. We briefly describe this model class for completeness, but refer readers to the literature for details [7, 6]. Regression trees are very similar to decision trees (which we used above for predicting satisfiability status). However, as a classification method, decision trees associate categorical labels with each leaf (e.g., "satisfiable"; "unsatisfiable"), while regression trees associate a real-valued prediction with each leaf. Random forests report an average over the predictions made by each of an ensemble of regression trees; these trees are made to differ by randomizing the training process.

Figure 4 illustrates the results of our broader experience with EHMs by highlighting three different solvers, each from a different domain. In each case we give plots for both quadratic ridge regression and random forests to show the impact of the learning algorithm. First (column 1), we considered the prominent SAT solver `Minisat 2.0` [9] running on a very heterogeneous mix of instances from the international SAT competition. Whereas the competition subdivides instances into categories ("industrial/application," "handmade/crafted," and "random"), we merged all instances together. Likely because of the heterogeneity of the resulting set, quadratic regression performed relatively poorly here. Random forests yielded much more reliable estimates; notably, they can partition the feature space into qualitatively different parts, and they never predict runtimes larger or smaller than the extrema observed in the training data. However, observe that even the less accurate quadratic regression models were usually accurate enough to differentiate between fast and slow runs in this domain; see the sidebar on `SATzilla`. Second (column 2), we studied the performance of the leading complete TSP solver, `Concorde` [2], on a widely used suite of rather homogeneous, randomly generated TSP instances [23]. We again see good performance, now for both quadratic regression and random forests. Third (columns 3 and 4), to show the effect of changing only the instance distribution, we consider one solver on two different distributions. IBM ILOG `CPLEX` [22] is the most widely used commercial MIP solver. BIGMIX is a highly heterogenous mix of publicly available mixed integer programming problems. As with the mix of SAT instances in our first benchmark, linear regression struggled with predictions for some types of instances and occasionally made catastrophic mispredictions. Again, random forests performed much more robustly. RCW models the dispersal and territory establishment of the red-cockaded woodpecker conditional on decisions about which parcels of land to protect [1]. The runtime of `CPLEX` for this domain was surprisingly predictable; random forests yielded among the best EHM performance we have ever observed.

Beyond Single Algorithms

Unlike average-case complexity results that characterize the inherent complexity of a computational problem, EHMs always describe the performance of a given algorithm. In some sense this is an inherent limitation: a statistical approach cannot summarize the performance of algorithms that have not yet been invented. However, there is a

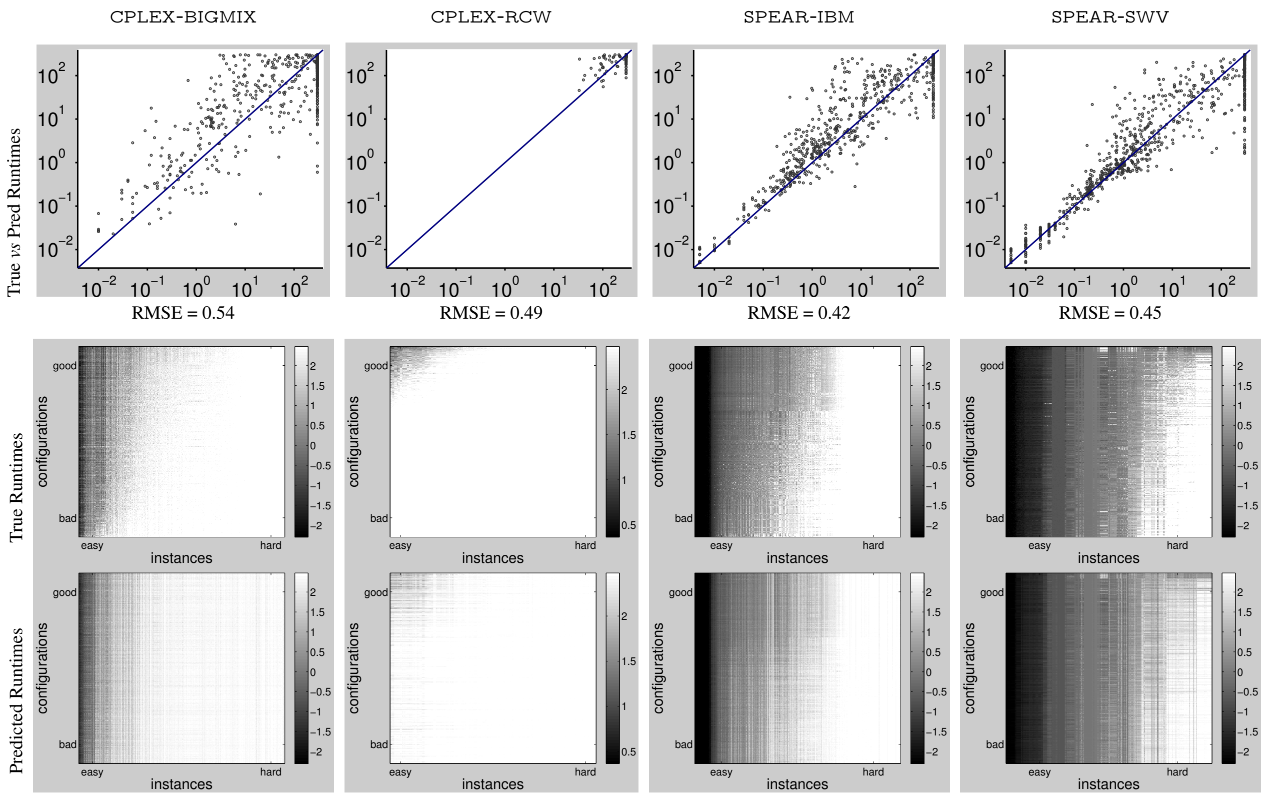

Figure 5: Visual comparison of models for runtime predictions on pairs of previously unseen test configurations and instances. Row 1: In each plot, the x-axis denotes true runtime and the y-axis denotes runtime as predicted by the respective model. Each dot represents one combination of an instance and parameter configuration, both previously unseen during model training. Rows 2 and 3: Actual and predicted runtimes for each application domain. Each dot in the heat map represents the run of one parameter configuration on one instance; the grayscale value represents runtime on a $\log_{10}$ scale (darker means faster).

useful way in which we can relax the single-algorithm restriction: we can build a model that describes a *space* of existing algorithms.

More specifically, most state-of-the-art algorithms for hard combinatorial problems offer a range of algorithm parameters in order to enable users to customize or tune the algorithm's behaviour. We define "parameters" very broadly, as encompassing any argument passed to a solver that changes its behaviour (and, thus, its runtime) but not the nature of the solution it returns. Parameters can thus be continuous, categorical, ordinal, or Boolean, and can even be conditional on values taken by other parameters. Importantly, categorical and Boolean parameters can be used to represent very abstract decisions—effectively selecting among unrelated blocks of code—and can thereby open up vast algorithm design spaces. For example, IBM ILOG CPLEX exposes 76 parameters (45 categorical, 6 Boolean, 18 integer, and 7 real-valued) [17]; a fairly coarse discretization of these parameters yields over 10^{47} different algorithm instantiations with vastly different performance profiles. We call such an instantiation of all parameters of a given algorithm to specific values a *configuration*. The second example of a parameterized solver we use here is the SAT solver SPEAR [4], which exposes 26 parameters (7 categorical, 3 Boolean, 4 integer, and 12 real-valued), giving rise to over 10^{17} different algorithm instantiations.

We now consider generalizing EHMs to describe such parameterized algorithms. In principle, this is not much of a change: we consider models that map from a joint space of configurations and instance features to runtime predictions. The question is how well such an approach can work. Before we can give an answer, we need to decide how to evaluate our methods. We could test on the same configurations that we used to train the EHM but on new problem instances, on new configurations but on previously seen instances, or on combinations of previously unseen configurations and instances. The third case is the hardest; it is the only setting for which we show results here. Figure 5 illustrates some representative results in this setting, focusing on random forest models. The first row shows scatterplots like those presented earlier, with each point representing a run of a randomly selected, previously unseen configuration on a previously unseen instance.We also provide a different way of looking at the joint space of parameter configurations and instance feature vectors. The second row shows the true runtimes for each (configuration, instance) pair, sorting configurations by their average performance and sorting instances by their average hardness. Thus, the picture gives a snapshot of the runtime variation across both instances and configurations, and makes it possible to gauge how much of this variation is due only to differences between configurations *vs* differences between instances. Finally, the third row shows the predictions obtained from the EHM in the same format as the second row. This gives a way of visually comparing model performance to ground truth; ideally, the second and third rows would look identical. (Indeed, when the two figures closely resemble each other, the EHM can serve as a *surrogate* for the original algorithm, meaning that the EHM can be substituted for the algorithm in an empirical analysis of the algorithm's performance; see [18].)

Application: Generating Hard Benchmarks

Realistic, hard benchmark distributions are important because they are used as an objective measure of success in algorithm development. However, it can sometimes be just as difficult to find new, hard benchmarks as it is to find new strategies for solving previously hard benchmarks. To fill this gap, EHMs can be used to automatically adjust existing instance generators so that they produce instances that are harder for a given set of algorithms [26, 29].

We start with an instance generator that has parameters p. Such generators are often simply used with default parameter settings; however, to search for harder instances we instead sample each parameter's value uniformly from a fixed range. Call the resulting distribution over instances $\mathcal{D}$. Our goal is to sample from a new distribution $\mathcal{D}'$ over the same instances that weights instances by their hardness for a given algorithm A. (Think of A as having the best average runtime among all algorithms in a given set, or as being a SATzilla-style selector among such algorithms.) We can do this via a form of importance sampling. We construct an EHM for A on $\mathcal{D}$ using our standard instance features f; for an instance x, call this model's prediction $H_f(x)$. We would like to generate a large set of instances from $\mathcal{D}$, weight each instance x in proportion to $H_f(x)$, and then sample a single instance from the set in proportion to the weights. This approach works, but requires a very large number of samples when hard instances are rare in $\mathcal{D}$. To improve performance, we learn a quadratic EHM H_p that uses only the generator parameters p as features. We can then sample instances x in proportion to $\mathcal{D}(x) \cdot H_p(x)$ rather than sampling from $\mathcal{D}$ (by sampling directly from polynomial function H_p, and then running the instance generator with the resulting parameters), and then weight each sampled instance x by $H_f(x)/H_p(x)$. H_p thus guides our search towards harder instances without biasing the weights. In experiments with the Combinatorial Auction Test Suite [30] this approach increased the mean hardness of generated instances by up to a factor of 100 [26, 29], and often created instances much harder than we had ever observed using the generators' default parameters.

The main takeaway from our experiments is that our models were able to achieve high accuracies (RMSEs around 0.5; qualitative similarity between the second and third rows) even on algorithm configurations that were never examined during training. The first column of Figure 5 concerns runtime predictions for CPLEX on our heterogeneous mix of MIP instances. The instances in this benchmark differ greatly in hardness, much more than the different CPLEX configurations differ in performance (see Row 2). As a result, instance hardness dominated runtimes and the model focused on the (more important) feature space, at the expense of failing to capture some of the performance difference between configurations. Second, on RCW most of the (randomly sampled) CPLEX configurations solved very few instances; we recorded such failures as very long runtimes. The model was nevertheless able to identify which configurations were good and which instances were easy. Finally, we consider predicting the runtime of SPEAR on two sets of formal verification instances: IBMis a set of bounded model checking instances [42], while SWVis a set of software verification instances generated with the Calysto static checker [3]. For both of these instance distributions, the runtime of SPEAR with different configurations was predicted with a high degree of accuracy. Our random forest models accurately predicted the empirical hardness of instances, the empirical performance of configurations, and even captured ways in which the two interact.

Application: Algorithm Configuration

Imagine that each time the designer of a heuristic algorithm faced a choice about a given design element, she simply encoded it as a free parameter of a single solver. In the end, her problem of designing a good algorithm for a given problem domain would be reduced to the stochastic optimization problem of finding a configuration that achieved good performance [24, 14, 12]. We have applied automated methods for solving this problem to identify novel algorithm configurations that have yielded orders of magnitude speedups in a broad range of domains, including SAT-based formal verification [15], MIP solving [17], and automated planning [37]. One state-of-the-art method for solving this problem is based on EHMs: *sequential model-based algorithm configuration (*SMAC*)* [19] iterates between (1) using an EHM to select promising configurations to explore next, (2) executing the algorithm with these configurations, and (3) updating the model with the resulting information. EHMs can also be used to select configurations on a per-instance basis [16].

Take-Away Messages

Statistical methods can characterize the difficulty of solving instances from a given distribution using the best available algorithms—even when those algorithms are extremely complex and traditional theoretical analysis is infeasible. Such *empirical hardness models* are surprisingly effective in practice, across different hard combinatorial problems, real-world instance distributions, and state-of-the-art solvers. An analysis of these models can serve as a starting point for new theoretical investigations into complexity beyond the worst case, by identifying problem features that are predictive of hardness or that suffice to predict an objective function (*e.g.*, satisfiability status) directly. In the context of highly parameterized algorithms that span a large space of possible algorithm designs, we have found that it is even possible to predict the runtime of previously untested algorithm designs on previously unseen instances. Empirical hardness models have proven useful in a variety of practical applications, including the automatic design of algorithm portfolios, the automatic synthesis of hard benchmark distributions, and the automatic search for a performance-optimizing design in a large algorithm design space. We have written open-source software for building EHMs, analyzing them, constructing algorithm portfolios, automatically configuring parameterized algorithms, and more: see http://www.cs.ubc.ca/labs/beta/Projects/Empirical-Hardness-Models/.

1. REFERENCES

[1] K. Ahmadizadeh, B. Dilkina, C.P. Gomes, and A. Sabharwal. An empirical study of optimization for maximizing diffusion in networks. In *Principles and Practice of Constraint Programming (CP'10)*, pages 514–521, 2010.

[2] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[3] D. Babić and A. J. Hu. Structural Abstraction of Software Verification Conditions. In *Computer Aided Verification (CAV'07)*, pages 366–378, 2007.

[4] D. Babić and F. Hutter. Spear theorem prover. Solver description, 2007 SAT Competition, 2007.

[5] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[6] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[7] L. Breiman, J. H. Friedman, R. Olshen, and C. J. Stone.

Classification and Regression Trees. Wadsworth, Belmont, California, 1984.

[8] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *International Joint Conference on Artificial Intelligence (IJCAI'91)*, pages 331–337, 1991.

[9] N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT'04)*, pages 502–518, 2004.

[10] J. Friedman. Multivariate adaptive regression splines. *Annals of Statistics*, 19(1):1–141, 1991.

[11] M. Heule and H. v. Maaren. march_hi. Solver description, SAT competition 2009, 2009.

[12] H.H. Hoos. Programming by optimisation. *Communications of the ACM*, 55(2):70–80, February 2012.

[13] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation*. Pearson Education, 2007.

[14] F. Hutter. *Automated Configuration of Algorithms for Solving Hard Computational Problems*. PhD thesis, University Of British Columbia, Department of Computer Science, Vancouver, Canada, October 2009.

[15] F. Hutter, D. Babić, H.H. Hoos, and A. J. Hu. Boosting Verification by Automatic Tuning of Decision Procedures. In *Conference on Formal Methods in Computer-Aided Design (FMCAD'07)*, pages 27–34, 2007.

[16] F. Hutter, Y. Hamadi, H.H. Hoos, and K. Leyton-Brown. Performance prediction and automated tuning of randomized and parametric algorithms. In *Principles and Practice of Constraint Programming (CP'06)*, pages 213–228, 2006.

[17] F. Hutter, H.H. Hoos, and K. Leyton-Brown. Automated configuration of mixed integer programming solvers. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'10)*, pages 186–202, 2010.

[18] F. Hutter, H.H. Hoos, and K. Leyton-Brown. Tradeoffs in the empirical evaluation of competing algorithm designs. *Annals of Mathematics and Artificial Intelligence*, 60(1):65–89, 2010.

[19] F. Hutter, H.H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization Conference (LION'11)*, pages 507–523, 2011.

[20] F. Hutter, D. A. D. Tompkins, and H.H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Principles and Practice of Constraint Programming (CP'02)*, pages 233–248, 2002.

[21] F. Hutter, L. Xu, H.H. Hoos, and K. Leyton-Brown. Algorithm runtime prediction: the state of the art, 2013. Accepted for publication in Artificial Intelligence; submitted version available as arXiv:1211.0906.

[22] IBM. IBM ILOG CPLEX Optimizer – Data Sheet. Available online: ftp://public.dhe.ibm.com/common/ssi/ecm/en/wsd14044usen/WSD14044USEN.PDF, 2011. Version last visited on January 26, 2012.

[23] D. S. Johnson. Random TSP generators for the DIMACS TSP Challenge. http://www2.research.att.com/~dsj/chtsp/codes.tar, 2011. Version last visited on May 16, 2011.

[24] A. KhudaBukhsh, L. Xu, H.H. Hoos, and K. Leyton-Brown. SATenstein: Automatically building local search SAT solvers from components. In *International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 517–524, 2009.

[25] D. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, 1975.

[26] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. Boosting as a metaphor for algorithm design. In *Principles and Practice of Constraint Programming (CP'03)*, pages 899–903, 2003.

[27] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. A portfolio approach to algorithm selection. In *International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 1542–1543, 2003.

[28] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *Principles and Practice of Constraint Programming (CP'02)*, pages 556–572, 2002.

[29] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*, 56(4):1–52, 2009.

[30] K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *ACM Conference on Electronic Commerce (ACM-EC'00)*, pages 66–76, 2000.

[31] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Conference on Artificial Intelligence (AAAI'92)*, pages 459–465, 1992.

[32] E. Nudelman, K. Leyton-Brown, G. Andrew, C. Gomes, J. McFadden, B. Selman, and Y. Shoham. Satzilla 0.9. Solver description, 2003 SAT Competition, 2003.

[33] E. Nudelman, K. Leyton-Brown, H.H. Hoos, A. Devkar, and Y. Shoham. Understanding random SAT: Beyond the clauses-to-variables ratio. In *Principles and Practice of Constraint Programming (CP'04)*, pages 438–452, 2004.

[34] M.R. Prasad, A. Biere, and A. Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer*, 7(2):156–173, 2005.

[35] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[36] B. Selman, H. J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Conference on Artificial Intelligence (AAAI'92)*, pages 440–446, 1992.

[37] M. Vallati, C. Fawcett, A. E. Gerevini, H.H. Hoos, and A. Saetti. Generating fast domain-optimized planners by automatically configuring a generic parameterised planner. In *Automated Planning and Scheduling Workshop on Planning and Learning (ICAPS-PAL'11)*, pages 21–27, 2011.

[38] L. Xu, H.H. Hoos, and K. Leyton-Brown. Hierarchical hardness models for SAT. In *Principles and Practice of Constraint Programming (CP'07)*, pages 696–711, 2007.

[39] L. Xu, H.H. Hoos, and K. Leyton-Brown. Predicting satisfiability at the phase transition. In *Conference on Artificial Intelligence (AAAI'12)*, pages 584 – 590, 2012.

[40] L. Xu, F. Hutter, H.H. Hoos, and K. Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, June 2008.

[41] L. Xu, F. Hutter, H.H. Hoos, and K. Leyton-Brown. Evaluating component solver contributions to portfolio-based algorithm selectors. In *Theory and Applications of Satisfiability Testing (SAT'12)*, pages 228–241, 2012.

[42] E. Zarpas. Benchmarking SAT Solvers for Bounded Model Checking. In *Theory and Applications of Satisfiability Testing (SAT'05)*, pages 340–354, 2005.