Understanding Random SAT: Beyond the Clauses-to-Variables Ratio

Eugene Nudelman[1], Alex Devkar[1], Yoav Shoham[1], and Kevin Leyton-Brown[2]

[1] Computer Science Department, Stanford University,Stanford, CA
{eugnud,avd,shoham}@cs.stanford.edu
[2] Computer Science Department, University of British Columbia, Vancouver, BC
kevinlb@cs.ubc.ca

Abstract. It is well known that the ratio of the number of clauses to the number of variables in a random k-SAT instance is highly correlated with the instance's empirical hardness. We consider the problem of identifying such features of random SAT instances automatically with machine learning. We describe and analyze models for three SAT solvers—kcnfs, oksolver and satz—and for two different distributions of instances: uniform random 3-SAT with varying ratio of clauses-to-variables, and uniform random 3-SAT with fixed ratio of clauses-to-variables. We show that surprisingly accurate models can be built in all cases. Furthermore, we analyze these models to determine which features are most useful in predicting whether an instance will be hard to solve. Finally we discuss other applications of our models including SATzilla, a portfolio of existing SAT solvers, which competed in the 2003 and 2004 SAT competitions.[3]

1 Introduction

SAT is among the most studied problems in computer science, representing a generic constraint satisfaction problem with binary variables and arbitrary constraints. It is also the prototypical $\mathcal{NP}$-Hard problem, and so its worst-case complexity has received much attention. Accordingly, it is not surprising that SAT has become a primary platform for the investigation of average-case and empirical complexity. Particular interest has been shown for randomly-generated SAT instances:this testbed offers a range of very easy to very hard instances for any given input size, yet the simplicity of the algorithm used to generate such instances makes them easier to understand analytically. Moreover, working on this testbed offers the opportunity to make connections to a wealth of existing work.

A seminal paper by Selman, Mitchell and Levesque [13] considered the empirical performance of DPLL-type solvers running on uniform-random k-SAT instances.[4] It found a strong correlation between the instance's hardness and the ratio of the number of clauses to the number of variables in the instance. Furthermore, it demonstrated

[3] We'd like to acknowledge very helpful assistance from Holger Hoos and Nando De Freitas, and our indebtedness to the authors of the algorithms in the SATzilla portfolio.

[4] Similar, contemporaneous work on phase transition phenomena in other hard problems was performed by Cheeseman [4], among others.

that the hardest region (*e.g.*, for random 3-SAT, a clauses-to-variables ratio of roughly 4.26) corresponds exactly to a phase transition in a non-algorithm-specific theoretical property of the instance: the probability that a randomly-generated formula having a given ratio will be satisfiable. This well-publicized finding led to increased enthusiasm for the idea of studying algorithm performance experimentally, using the same tools as are used to study natural phenomena. Over the past decade, this approach has complemented more traditional theoretical worst-case analysis of algorithms, with interesting findings on (*e.g.*) islands of tractability [7], search space topologies for stochastic local search algorithms [6], backbones [12], backdoors [16] and random restarts [5] that have improved our understanding of algorithms' empirical behavior.

Inspired by the success of this work in SAT and related problems, in 2001 we proposed a new methodology for using machine learning to study empirical hardness [10]. We applied this methodology to the Combinatorial Auction Winner Determination Problem (WDP)—an $\mathcal{NP}$-hard combinatorial optimization problem equivalent to weighted set packing. In later work [9, 8] we extended our methodology, demonstrating techniques for improving empirical algorithm performance through the construction of algorithm portfolios, and for automatically inducing hard benchmark distributions. In this paper we come full circle and apply our methodology to SAT—the original inspiration for its development.

This work has three goals. Most directly, it aims to show that inexpensively-computable features can be used to make accurate predictions about the empirical hardness of random SAT instances, and to analyze these models in order to identify important features. We consider three different SAT algorithms (`kcnfs`, `oksolver` and `satz`, each of which performed well in the Random category in one or more past SAT competitions) and two different instance distributions. The first instance distribution, random 3-SAT instances where the ratio of clauses-to-variables is drawn uniformly from $[3.26, 5.26]$, allows us to find out whether our techniques would be able to automatically discover the importance of the clauses-to-variables ratio in a setting where it is known to be important, and also to investigate the importance of other features in this setting. Our second distribution is uniform-random 3-SAT with the ratio of clauses-to-variables held constant at the phase transition point of 4.26. This distribution has received much attention in the past, and poses an interesting puzzle: orders-of-magnitude runtime variation persists in this so-called "hard region."

Second, we show that empirical hardness models have other useful applications for SAT. Most importantly, we describe a SAT solver, `SATzilla`, which uses hardness models to choose among existing SAT solvers on a per-instance basis. We explain some details of its construction and summarize its performance in the 2003 SAT competition.

Our final goal is to offer a concrete example in support of our abstract claim that empirical hardness models are a useful tool for gaining understanding about the behavior of algorithms for solving $\mathcal{NP}$-hard problems. Thus, while we believe that our SAT results are interesting in their own right, it is important to emphasize that very few of our techniques are particular to SAT, and indeed that we have achieved equally strong results when applying our methodologies to other, qualitatively different problems.[5]

[5] WDP is a very different problem from SAT: feasible solutions for WDP can be identified in constant time, and the goal is to find an *optimal* feasible solution. There is thus no opportunity

2 Methodology

Although the work surveyed above has led to great advances in understanding the empirical hardness of SAT problems, most of these approaches scale poorly to more complicated domains. In particular, most of these methods involve exhaustive exploration of the search and/or distribution parameter spaces, and require considerable human intervention and decision-making. As the space of relevant features grows and instance distributions become more complex, it is increasingly difficult either to characterize the problem theoretically or to explore its degrees of freedom exhaustively. Moreover, most current work focuses on understanding algorithms' performance profiles, rather than trying to characterize the hardness of individual problem instances.

2.1 Empirical Hardness Models

In [10] we proposed a novel experimental approach for predicting the runtime of a given algorithm on individual problem instances:

1. **Select a problem instance distribution.**
   Observe that since we are interested in the investigation of *empirical* hardness, the choice of distribution is fundamental—different distributions can induce very different algorithm behavior. It is convenient (though not necessary) for the distribution to come as a parameterized generator; in this case, a distribution must be established over the generator's parameters.
2. **Select one or more algorithms.**
3. **Select a set of inexpensive, distribution-independent features.**
   It is important to remember that individual features need not be perfectly predictive of hardness; ultimately, our goal will be to combine features together. The process of identifying features relies on domain knowledge; however, it is possible to take an inclusive approach, adding all features that seem reasonable and then removing those that turned out to be unhelpful (see step 5). It should be noted, furthermore, that many features that proved to be useful for one constraint problem can carry over into another.
4. **Sample the instance distribution to generate a set of instances. For each instance, determine the running time of the selected algorithms and compute the features.**
5. **Eliminate redundant or uninformative features.**
   As a practical matter, much better models tend to be learned when all features are informative. A variety of statistical techniques are available for eliminating or deemphasizing the effect of such features. The simplest one is to manually examine pairwise correlations, eliminating features that are highly correlated with what

to terminate the algorithm the moment a solution is found, as in SAT. While algorithms for WDP usually find the optimal solution quickly, they spend most of their time proving optimality, a process analogous to proving unsatisfiability. We also have unpublished initial results showing promising hardness model performance for TSP and computation of Nash equilibria.

remains. Shrinkage techniques (such as lasso [14] or ridge regression) are another alternative.

6. **Use machine learning to select a function of the features that predicts each algorithm's running time.**
   Since running time is a continuous variable, regression is the natural machine-learning approach to use for building runtime models. (For more detail about why we prefer regression to other approaches such as classification, see [10].) We describe the model-construction process in more detail in the next section.

2.2 Building Models

There are a wide variety of different regression techniques; the most appropriate for our purposes perform supervised learning.[6] Such techniques choose a function from a given hypothesis space (*i.e.*, a space of candidate mappings from the given features to the running time) in order to minimize a given error metric (a function that scores the quality of a given mapping, based on the difference between predicted and actual running times on training data, and possibly also based on other properties of the mapping). Our task in applying regression to the construction of hardness models thus reduces to choosing a hypothesis space that is able to express the relationship between our features and our response variable (running time), and choosing an error metric that both leads us to choose good mappings from this hypothesis space and can be tractably minimized.

The simplest regression technique is linear regression, which learns functions of the form $\sum_i w_i f_i$, where f_i is the i^{th} feature and the w's are free variables, and has as its error metric root mean squared error. Linear regression is a computationally appealing procedure because it reduces to the (roughly) cubic-time problem of matrix inversion. In comparison, most other regression techniques depend on more difficult optimization problems such as quadratic programming.

Choosing an Error Metric Linear regression uses a squared-error metric, which corresponds to the L_2 distance between a point and the learned hyperplane. Because this measure penalizes outlying points superlinearly, it can be inappropriate in cases where data contains many outliers. Some regression techniques use L_1 error (which penalizes outliers linearly); however, optimizing these error metrics often requires solution of a quadratic programming problem.

Some error metrics express an additional preference for models with small (or even zero) coefficients to models with large coefficients. This can lead to much more reliable models on test data, particularly in cases where features are correlated. Some examples of such "shrinkage" techniques are ridge, lasso and stepwise regression. Shrinkage techniques generally have a parameter that expresses the desired tradeoff between training error and shrinkage; this parameter is generally tuned using either cross-validation or a validation set.

[6] Because of our interests in being able to analyze our models and in keeping model sizes small (*e.g.*, so that models can be distributed as part of an algorithm portfolio), we avoid model-free approaches such as nearest neighbor.

Choosing a Hypothesis Space Although linear regression seems quite limited, it can actually be used to perform regression in a wide range of hypothesis spaces. There are two key tricks. The first is to introduce new features which are functions of the original features. For example, in order to learn a model which is a quadratic rather than a linear function of the features, the feature set can be augmented to include all pairwise products of features. A hyperplane in the resulting much-higher-dimensional space corresponds to a quadratic manifold in the original feature space. The key problem with this approach is that the set of features grows quadratically, which may cause the regression problem to become intractable (*e.g.*, because the feature matrix cannot fit into memory) and can also lead to overfitting (when the hypothesis space becomes expressive enough to fit noise in the training data). In this case, it can make sense to add only a subset of the pairwise products of features; *e.g.*, one heuristic is to add only pairwise products of the k most important features in the linear regression model. Of course, we can use the same idea to reduce many other nonlinear hypothesis spaces to linear regression: all hypothesis spaces which can be expressed by $\sum_i w_i g_i(\mathbf{f})$, where g_i is an arbitrary function and $\mathbf{f}$ is the set of all features.

Sometimes we want to consider hypothesis spaces of the form $h\left(\sum_i w_i g_i(\mathbf{f})\right)$. For example, we may want to fit a sigmoid or an exponential curve. When h is a one-to-one function, we can transform this problem to a linear regression problem by replacing our response variable y in our training data by $h^{-1}(y)$, where h^{-1} is the inverse of h, and then training a model of the form $\sum_i w_i g_i(\mathbf{f})$. On test data, we must evaluate the model $h\left(\sum_i w_i g_i(\mathbf{f})\right)$. One caveat about this trick is that it distorts the error metric: the error-minimizing model in the transformed space will not generally be the error-minimizing model in the true space. In many cases this distortion is acceptable, however, making this trick a tractable way of performing many different varieties of nonlinear regression.

Two examples that we will discuss later in the paper are exponential models ($h(x) = 10^x$; $h^{-1}(x) = \log_{10}(x)$) and logistic models ($h(x) = 1/(1+e^{-x})$; $h^{-1}(x) = \ln(x)\ln(1-x)$ with values of x first mapped on to the interval $(0,1)$). Because they evaluate to values on a finite interval, we have found logistic models to be particularly useful in cases where runs were capped.

2.3 Evaluating the Importance of Variables in a Hardness Model

If we are able to construct an accurate empirical hardness model, it is natural to try to explain why it works. A key question is which features were most important to the success of the model. It is tempting to interpret a linear regression model by comparing the coefficients assigned to the different features, on the principle that larger coefficients indicate greater importance. This can be misleading for two reasons. First, features may have different ranges, a problem that can be addressed by normalization. A more fundamental problem arises in the presence of correlated features. For example, if two features are unimportant but perfectly correlated, they could appear in the model with arbitrarily large coefficients but opposite signs. A better approach is to force models to contain fewer variables, on the principle that the best low-dimensional model will involve only relatively uncorrelated features. Once such a model has been obtained, we can evaluate the importance of each feature to that model by looking at each feature's *cost of omission*. That is, we can train a model without the given feature and report the

resulting increase in (cross-validated) prediction error. To make them easier to compare, we scale the cost of omission of the most important feature to 100 and scale the other costs of omission in proportion.

There are many different "subset selection" techniques for finding good, small models. Ideally, exhaustive enumeration would be used to find the best subset of features of desired size. Unfortunately, this process requires consideration of a binomial number of subsets, making it infeasible unless both the desired subset size and the number of base features are very small. When exhaustive search is impossible, heuristic search can still find good subsets. The most well-known ones heuristics are *forward selection*, *backward elimination* and *sequential replacements*. Forward selection starts with an empty set, and greedily adds the feature that, combined with the current model, makes the largest reduction to cross-validated error. Backward elimination starts with a full model and greedily removes the features that yields the smallest increase in cross-validated error. Sequential replacement is like forward selection, but also has the option to replace a feature in the current model with an unused feature. Finally, the recently introduced LAR [2] algorithm is a shrinkage technique for linear regression that can set the coefficients of sufficiently unimportant variables to zero as well as simply reducing them; thus, it can be also be used for subset selection. Since none of these four techniques is guaranteed to find the optimal subset, we combine them together by running all four and keeping the model with the smallest cross-validated (or validation-set) error.

3 Hardness Models for SAT

3.1 Features

Fig. 1 summarizes the 91 features used by our SAT models. However, these features are not all useful for every distribution: as we described above, we eliminate uninformative or highly correlated features after fixing the distribution. For example, while ratio of clauses-to-variables was important for `SATzilla`, it is not at all useful for the dataset that studies solvers performance at a fixed ratio at a phase transition point. In order to keep values to sensible ranges, whenever it makes sense we normalize features by either the number of clauses or the number of variables in the formula.

The features can be roughly categorized into 9 groups. The first group captures problem size, measured by the number of clauses, variables, and the ratio of the two. Because we expect this ratio to be an important feature, we include squares and cubes of both the ratio and its reciprocal. Also, because we know that features are more powerful in simple regression models when they are directly correlated with the response variable, we include a "linearized" version of the ratio which is defined as the absolute value of the difference between the ratio and the phase transition point, 4.26. The next three groups correspond to three different graph representations of a SAT instance. Variable-Clause Graph (VCG) is a bipartite graph with a node for each variable, a node for each clause, and an edge between them whenever a variable occurs in a clause. Variable Graph (VG) has a node for each variable and an edge between variables that occur together in at least one clause. Clause Graph (CG) has nodes representing clauses and an edge between two clauses whenever they share a negated literal. All of these graphs correspond to constraint graphs for the associated CSP; thus, they encode the problem's

Problem Size Features:

1. **Number of Clauses:** denote this c
2. **Number of Variables:** denote this v

3-5. **Ratio:** c/v, $(c/v)^2$, $(c/v)^3$

6-8. **Ratio Reciprocal:** (v/c), $(v/c)^2$, $(v/c)^3$

9-11. **Linearized Ratio:** $|4.26 - c/v|$, $|4.26 - c/v|^2$, $|4.26 - c/v|^3$

Variable-Clause Graph Features:

12-16. **Variable nodes degree statistics**: mean, variation coefficient, min, max and entropy.

17-21. **Clause nodes degree statistics**: mean, variation coefficient, min, max and entropy.

Variable Graph Features:

22-25. **Nodes degree statistics**: mean, variation coefficient, min, and max.

Clause Graph Features:

26-32. **Nodes degree statistics**: mean, variation coefficient, min, max, and entropy.

33-35. **Clustering Coefficient Statistics**: mean, variation coefficient, min, max, and entropy.

Balance Features:

36-40. **Ratio of positive and negative literals in each clause**: mean, variation coefficient, min, max, and entropy.

41-45. **Ratio of positive and negative occurences of each variable**: mean, variation coefficient, min, max, and entropy.

46-48. **Fraction of unary, binary, and ternary clauses**

Proximity to Horn Formula

49. **Fraction of Horn Clauses**

50-54. **Number of occurences in a Horn Clause for each variable** : mean, variation coefficient, min, max, and entropy.

LP-Based Features:

55. **Objective value of LP relaxation**
56. **Fraction of variables set to 0 or 1**

57-60. **Variable integer slack statistics:** mean, variation coefficient, min, max.

DPLL Search Space:

61-65. **Number of Unit propagations:** computed at depths 1,4,16,64, and 256

66-67. **Search Space size estimate:** mean depth till contradiction, estimate of the log of number of nodes.

Local Search Probes:

68-71. **Minimum fraction of unsat clauses in a run:** mean and variation coefficient for SAPS and GSAT.

72-81. **Number of steps to the best local minimum in a run:** mean, median, variation coefficient, 10^{th} and 90^{th} percentiles for SAPS and GSAT.

82-85. **Average improvement to best:** For each run, we calculate the mean improvement per step to best solution. We then compute mean and variation coefficient over all runs for SAPS and GSAT.

86-89. **Fraction of improvement due to first local minimum:** mean and variation coefficient for SAPS and GSAT.

90-91. **Coefficient of variation of the number of unsatisfied clauses in each local minimum:** mean over all runs for SAPS and GSAT.

Fig. 1. SAT Instance Features

combinatorial structure. For all graphs we compute various node degree statistics. For CG we also compute statistics of clustering coefficients, which measure the extent to which each node belongs to a clique. For each node the clustering coefficient is the number of edges between its neighbors divided by $k(k-1)/2$, where k is the number of neighbors. The fifth group measures the balance of a formula in several different senses: we compute the number of unary, binary, and ternary clauses; statistics of the number of positive vs. negative occurrences of variables within clauses and per variable. The sixth group measures the proximity of the instance to a Horn formula, motivated by the fact that such formulas are an important SAT subclass. The seventh group of features is obtained by solving a linear programming relaxation of an integer program representing the current SAT instance. (In fact, on occasion this relaxation is able to solve the SAT instance!) Denote the formula $C_1 \wedge \ldots \wedge C_n$ and let x_j denote both boolean and LP variables. Define $v(x_j) = x_j$ and $v(\neg x_j) = 1 - x_j$. Then the program is maximize $\sum_{i=1}^{n} \sum_{l \in C_i} v(l)$ subject to $\forall C_i : \sum_{l \in C_i} v(l) \geq 1, \forall x_j : 0 \leq x_j \leq 1$. The objective function prevents the trivial solution where all variables are set to 0.5. The eighth group involves running DPLL "probes." First, we run a DPLL procedure (without backtracking) to an exponentially-increasing sequence of depths, measuring the number of unit propagations done at each depths. We also run depth-first random

probes by repeatedly instantiating random variables and performing unit propagation until a contradiction is found. The average depth at which a contradiction occurs is an unbiased estimate of the log size of the search space [11]. Our final group of features probes the search space with two stochastic local search algorithms, GSAT and SAPS. We run both algorithms many times, each time continuing the search trajectory until a plateau cannot be escaped within a given number of steps. We then average various statistics collected during each run.

3.2 Experimental Setup

Our first dataset contained 20000 uniformly-random 3-SAT instances with 400 variables each. To determine the number of clauses in each instance, we determined the clauses-to-variables ratio by drawing a uniform sample from $[3.26, 5.26]$ (*i.e.*, the number of clauses varied between 1304 and 2104). Our second dataset also contained 20000 fixed size 3-SAT instances. In this case each instance was generated uniformly at random with a fixed clauses-to-variables ratio of 4.26. We again generated 400-variable formulas; thus each formula had 1704 clauses. On each dataset we ran three solvers—`kcnfs`, `oksolver` and `satz`—which performed well on random instances in previous years' SAT competitions. Our experiments were executed on 2.4 GHz Xeon processors, under Linux 2.4.20. Our fixed-size experiments took about four CPU-months to complete. In contrast, our variable-size dataset took only about one CPU-month, since many instances were generated in the easy region away from the phase transition point. Every solver was allowed to run to completion on every instance.

Each dataset was split into 3 parts—training, test and validation sets—in the ratio $70 : 15 : 15$. All parameter tuning was performed with the validation set; the test set was used only to generate the graphs shown in this paper. We used the `R` and `Matlab` software packages to perform all machine learning and statistical analysis tasks.

4 Variable-Size Random Data

Our first set of experiments considered a set of uniform random 3-SAT instances where the clauses-to-variables ratio was drawn from the interval $[3.26, 5.26]$. We had three goals with this distribution. First, we wanted to show that our empirical hardness model training and analysis techniques would be able to automatically "discover" that the clauses-to-variables ratio was important to the empirical hardness of instances from this distribution. Second, having included nine features derived from this ratio among our 91 features—the clauses-to-variables ratio itself, the square of the ratio, the cube of the ratio, its reciprocal (*i.e.*, the variables-to-clauses ratio), the square and cube of this reciprocal, the absolute value minus 4.26, and the square and cube of this absolute value—we wanted to find out what particular function of these features would be most predictive of hardness. Third, we wanted to find out what *other* features, if any, would be important to a model in this setting.

It is worthwhile to start by examining the clauses-to-variables ratio in more detail. Fig. 2 shows `kcnfs` runtime (log scale) vs. c/v. First observe that, unsurprisingly, there is a clear relationship between runtime and c/v. At the same time, c/v is not a

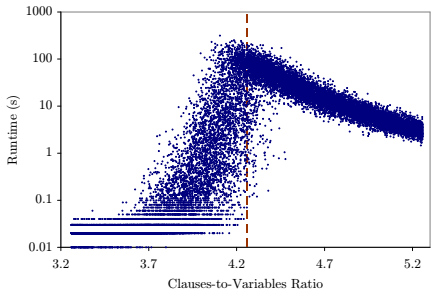

Fig. 2. Easy hard easy transition on variable-size data for `kcnfs`

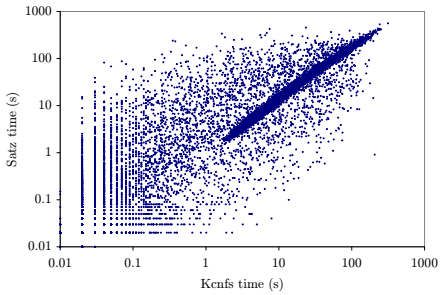

Fig. 3. `kcnfs` vs. `satz` on all instances

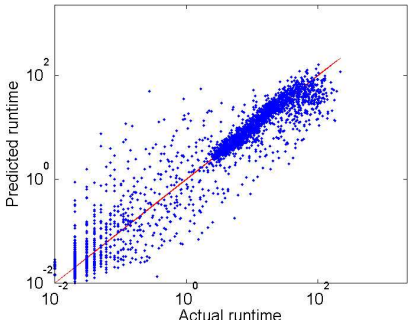

Fig. 4. VS Logistic model for `kcnfs`

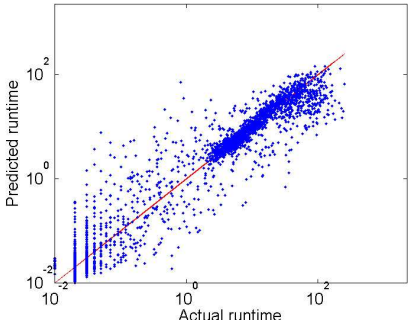

Fig. 5. VS Logistic model for `satz`

very accurate predictor of hardness by itself: particularly near the phase transition point, there are several orders of magnitude of runtime variance across different instances.

To build models, we first considered linear, logistic and exponential models in our 91 features, evaluating the models on our validation set. Of these, linear were the worst and logistic and exponential were similar, with logistic being slightly better. Next, we wanted to consider quadratic models under these same three transformations. However, a full quadratic model would have involved 4277 features, and given that our training data involved 14000 different problem instances, training the model would have entailed inverting a matrix of nearly sixty million values. In order to concentrate on the most important quadratic features, we first used our variable importance techniques to identify the best 30-feature subset of our 91 features. We computed the full quadratic expansion of these features, then performed forward selection—the only subset selection technique that worked with such a huge number of features—to keep only the most useful features. We ended up with 368 features, some of which were members of our original set of 91 features and the rest of which were products of these original features. Again, we evaluated linear, logistic and exponential models; all three model types were better with the expanded features, and again logistic models were best.

Figs 4 and 5 show our logistic models in this quadratic case for `kcnfs` and `satz` (both evaluated for the first time on our test set). First, note that these are incredibly accurate models: perfect predictions would lie exactly on the line $y = x$, and in these scatter-plots the vast majority of points like on or very close to this line, with no significant

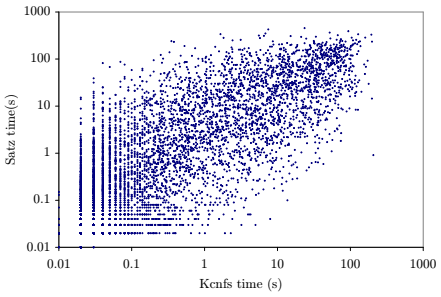

Fig. 6. `kcnfs` vs. `satz`, SAT instances

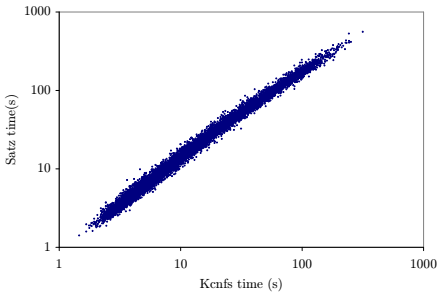

Fig. 7. `kcnfs` vs. `satz`, UNSAT instances

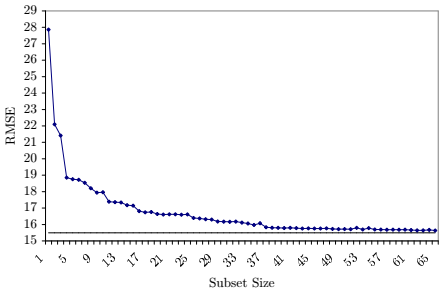

Fig. 8. VS `kcnfs` Subset Selection

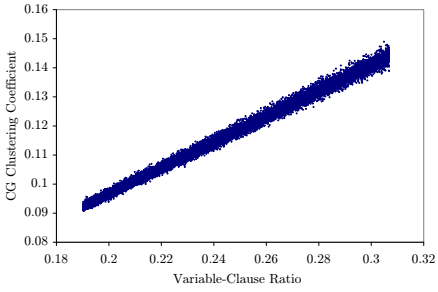

Fig. 9. CG Clustering Coefficient vs. v/c

bias in the residuals.[7] The RMSE for the `kcnfs`, `satz` and `oksolver` models are 16.02, 18.64 and 18.96 seconds respectively. Second, the scatterplots look very similar (as does the plot for `oksolver`, not shown here.)

The next natural question is whether this similar model performance occurs because runtimes of the two algorithms are strongly correlated. Fig. 3 shows `kcnfs` runtime vs. `satz` runtime on all instances. Observe that there appear to be two qualitatively different patterns in this scatterplot. We plotted satisfiable and unsatisfiable instances separately in Figs. 6 and 7, and indeed the different categories exhibit entirely different behavior: runtimes of unsatisfiable instances were almost perfectly correlated, while runtimes of satisfiable instances were almost entirely uncorrelated. We conjecture that this is because proving unsatisfiability of an instance requires exploring the whole search tree, which does not differ substantially between the algorithms, while finding a satisfiable assignment depends on each algorithm's different heuristics. We can conclude that the similarly accurate model performance between the algorithms is due jointly to the correlation between their runtimes on UNSAT instances and to the ability of our features to express both the runtimes of these UNSAT instances and the runtimes of each algorithm's runtime profile on SAT instances.

We now turn to the question of what variables were most important to our models. Because of space constraints, for the remainder of this paper we focus only on our

[7] The banding on very small runtimes in this and other scatterplots is a discretization effect due to the low resolution of the operating system's process timer.

Kcnfs	Cost of Omission
abs(CLAUSE_VARS_RATIO - 4.26) (9)	100
VARS_CLAUSES_RATIO_CUBE (8)	46
SAPS_BestStep_CoeffVar (74) × SAPS_BestCoeffVar_Mean (90)	41
GSAT_BestStep_Mean (77) × GSAT_AvgImproveToBest_Mean (84)	37

Table 1. Variable Importance in Variable Size Models

models for `kcnfs`.[8] Fig. 8 shows the validation set RMSE of our best subset of each size. Note that our best four-variable model achieves a root-mean-squared error of 19 seconds, while our full 368-feature model had an error of about 15.5 seconds. Table 4 lists the four variables in this model along with their normalized costs of omission. Note that our most important feature (by far) is the linearized version of c/v, and our second most important feature is $(v/c)^3$. This represents the satisfaction of our first and second objectives for this dataset: our techniques correctly identified the importance of the clauses-to-variables ratio and also informed us of which of our nine variants of this feature were most useful to our models.

The third and fourth variables in this model satisfy our third objective: we see that c/v variants are not the only useful features in this model. Interestingly, both of these remaining variables are constructed from local search probing features. It may be initially surprising that local search probes can convey meaningful information about the runtime behavior of DPLL searches. However, observe that the two approaches' search spaces and search strategies are closely related. Consider the local-search objective function "number of satisfied clauses." Non-backtrack steps in DPLL can be seen as monotonic improvements to this objective function in the space of partial truth assignments, with backtracking occurring only when no such improvement is possible. Furthermore, a partial truth assignment corresponds to a set of local search states, where each variable assignment halves the cardinality of this set and every backtrack doubles it. Since both search strategies alternate between monotonic improvements to the same objective functions and jumps to other (often nearby) parts of the search space, it is not surprising that large-scale topological features of the local search space are correlated with the runtimes of DPLL solvers. Since GSAT and SAPS explore the local search space very differently, their features give different but complementary views of the same search topology. Broadly speaking, GSAT goes downhill whenever it can, while SAPS uses its previous search experience to construct clause weights, which can sometimes influence it to move uphill even when a downhill option exists.

In passing, we point out an interesting puzzle that we encountered in analyzing our variable-size models. We discovered that the clause graph clustering coefficient is almost perfectly correlated with v/c, as illustrated in Fig. 9. This is particularly interesting as the clustering coefficient of the constraint graph has been shown to be an important statistic in a wide range of combinatorial problems. We haven't been able to to find any reference in the SAT literature relating CGCC to v/c. However, our intuition leads us to believe that this nearly-linear relationship should hold with high probability and that it will provide new insights into why v/c is correlated with hardness.

[8] We choose to focus on this algorithm because it is currently the state-of-the-art random solver; our results with the other two algorithms are comparable.

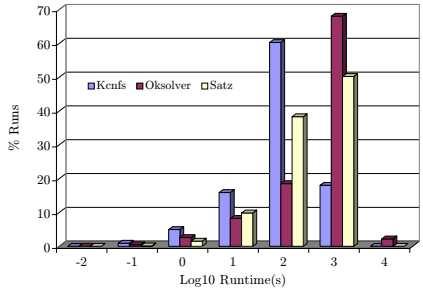

Fig. 10. FS Gross Hardness

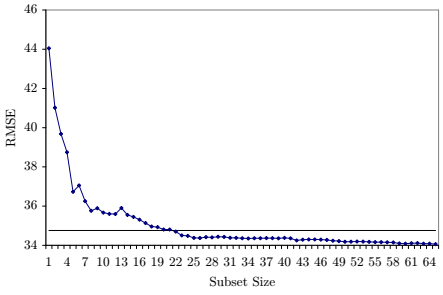

Fig. 11. FS `kcnfs` Subset Selection

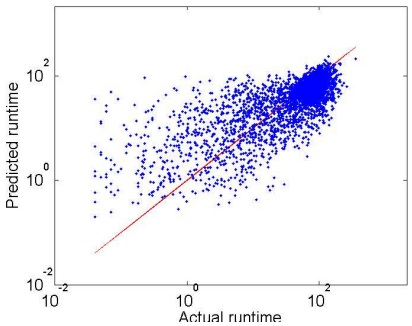

Fig. 12. FS `kcnfs` Logistic Model

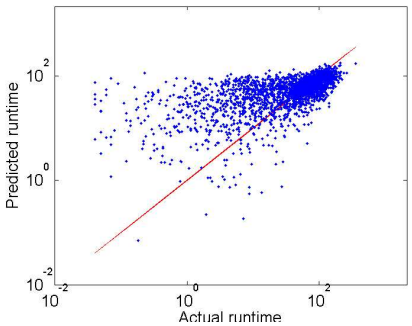

Fig. 13. FS `kcnfs` Linear Model

5 Fixed-Size Random Data

Conventional wisdom has it that uniform-random 3-SAT is easy when far from the phase-transition point, and hard when close to it. In fact, while the first part of this statement is generally true, the second part is not. Fig. 10 shows histograms of our three algorithms on our second dataset, fixed-size instances generated with $c/v = 4.26$. We can see that there is substantial runtime variation in this "hard" region: each algorithm is well represented in at least three of the order-of-magnitude bins. This distribution is an interesting setting for the construction of empirical hardness models—our most important features from the variable size distribution, variants of c/v, are constant in this case. We are thus forced to concentrate on new sources of hardness. This distribution is also interesting because, since the identification of the c/v phase transition, it has become perhaps the most widely used SAT benchmark.

We built models in the same way as described in Section 4, except that we omitted all variants of c/v because they were constant. Again, we achieved the best (validation set) results with logistic models on a (partial) quadratic expansion of the features. Fig. 12 shows the performance of our logistic model for `kcnfs` on test data. For the sake of comparison, Fig. 13 shows the performance of our linear model for `kcnfs` (again with the quadratic expansion of the features) on the same test data. Although both models are surprisingly good—especially given the difficulty of the dataset—the linear model shows considerably more bias in the residuals.

Kcnfs	Cost of Omission
SAPS_BestSolution_Mean (68) × GSAT_BestSolution_Mean (70)	100
SAPS_BestStep_Mean (72)× GSAT_BestSolution_Mean (70)	77
SAPS_BestSolution_Mean (68)× Mean_DPLL_Depth (66)	48
SAPS_BestSolution_CoeffVar (69) × SAPS_BestStep_Mean (72)	22
SAPS_BestSolution_CoeffVar (69) × SAPS_BestCoeffVar_Mean (90)	16
SAPS_BestSolution_Mean (68) × SAPS_BestStep_CoeffVar (74)	13
SAPS_BestStep_CoeffVar (74) × SAPS_BestCoeffVar_Mean (90)	8
CG_Degree_Mean (26) × SAPS_BestSolution_CoeffVar (69)	1

Table 2. Variable Importance in Fixed Size Models

Fig. 11 shows the validation set RMSE of the best model we found at each subset size. In this case we chose to study the 8-variable model. The variables in the model, along with their costs of omission, are given in Table 2. This time local search probing features are nearly dominant; this is particularly interesting since the same features appear for both local search algorithms, and since most of the available features never appear. The most important local search concepts appear to be the objective function value at the deepest plateau reached on a trajectory (BestSolution), and the number of steps required to reach this deepest plateau (BestStep).

6 SATzilla and Other Applications of Hardness Models

While the bulk of this paper has aimed to show that accurate empirical hardness models are useful because of the insight they give into problem structure, these models also have other applications [8]. For example, it is very easy to combine accurate hardness models with an existing instance generator to create a new generator that makes harder instances, through the use of rejection sampling techniques. Within the next few months, we intend to make available a new generator of harder random 3-SAT formulas. This generator will work by generating an instance from the phase transition region and then rejecting it in inverse proportion to the log time of the minimum of our three algorithms' predicted runtimes.

A second application of hardness models is the construction of algorithm portfolios. It is well known that for SAT (as for many other hard problems) different algorithms often perform very differently on the same instances. Indeed, this is very clear in Fig. 6, which shows that `kcnfs` and `satz` are almost entirely uncorrelated in their runtimes on satisfiable random 3-SAT instances. On distributions for which this sort of uncorrelation holds, selecting an algorithm to run on a per-instance basis offers the potential for substantial improvements over per-distribution algorithm selection. Empirical hardness models allow us to do just this, build algorithm portfolios that select an algorithm to run based on predicted runtimes.

We can offer concrete evidence for the utility of our this second application of hardness models: `SATzilla`, an algorithm portfolio that we built for the 2003 SAT competition [3]. This portfolio consisted of `2clseq`, `eqSatz`, `HeerHugo`, `JeruSat`, `Limmat`, `oksolver` , `Relsat`, `Sato`, `Satz-rand` and `zChaff`. At the time of writing, a second version of `SATzilla` is participating in the 2004 SAT competition. This version drops `HeerHugo`, but adds `Satzoo`, `kcnfs` , and `BerkMin`, new solvers that appeared in 2003 and performed well in the 2003 competition.

To construct `SATzilla` we began by assembling a library of about 5000 SAT instances, which we gathered from various public websites, for which we computed

runtimes and the features described in Section 3.1. We built models using ridge regression. To yield better models, we dropped from our dataset all instances that were solved by all algorithms, by no algorithms, or as a side-effect of feature computation.

Upon execution, SATzilla begins by running a UBCSAT [15] implementation of WalkSat for 30 seconds to filter out easy satisfiable instances. Next, SATzilla runs the Hypre[1] preprocessor to clean up instances, allowing the subsequent analysis of their structure to better reflect the problem's combinatorial "core."[9] Third, SATzilla computes its features, terminating if any feature (*e.g.*, probing; LP relaxation) solves the problem. Some features can also take an inordinate amount of time, particularly with very large inputs. To prevent feature computation from consuming all of our allotted time, certain features run only until a timeout is reached, at which point SATzilla gives up on computing the given feature. Fourth, SATzilla evaluates a hardness model for each algorithm. If some of the features have timed out, SATzilla uses a different model which does not involve the missing feature and which was trained only on instances where the same feature timed out. Finally, SATzilla executes the algorithm with the best predicted runtime. This algorithm continues to run until the instance is solved or until the allotted time is used up.

As described in the official report written by the 2003 SAT competition organizers [3], SATzilla's performance in this competition demonstrated the viability of our portfolio approach. SATzilla qualified to enter the final round in two out of three benchmark categories – Random and Handmade. Unfortunately, a bug caused SATzilla to crash often on Industrial instances (due to their extremely large size) and so SATzilla did not qualify for the final round in this category. During the competition, instances were partitioned into different *series* based on their similarity. Solvers were then ranked by the number of series in which they managed to solve at least one benchmark. SATzilla placed second in the Random category (the first solver was kcnfs, which wasn't in the portfolio as it hadn't yet been publicly released). In the Handmade instaces category SATzilla was third (2^{nd} on satisfiable instances), again losing only to new solvers.

Figures 14 and 15 show the raw number of instances solved by the top four finalists in each of the Random and Handmade categories, in both cases also including the top-four solvers from the other category which qualified. In general the solvers that did well in one category did very poorly (or didn't qualify for the final) in the other. SATzilla is the only solver which achieved strong performance in both categories.

During the 2003 competition, we were allowed to enter a slightly improved version of SATzilla that was run as an *hors concours* solver, and thus was not run in the finals. According to the competition report, this improved version was first in the Random instances category both in the number of actual instances solved, and in the total runtime used (though still not in the number of series solved). As a final note, we should point out that the total development time for SATzilla was under a month—considerably less than most world-class solvers, though of course SATzilla relies on the existence of base solvers.

[9] Despite the fact that this step led to more accurate models, we did not perform it in our investigation of uniform-random 3-SAT because it implicitly changes the instance distribution. Thus, while our models would have been more accurate, they would also have been less informative.

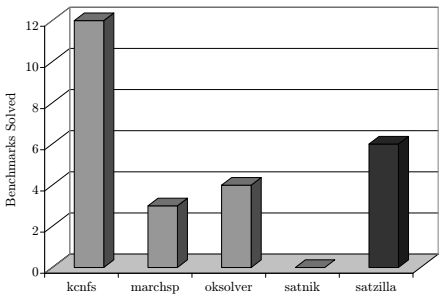

Fig. 14. SAT-2003 Random Category

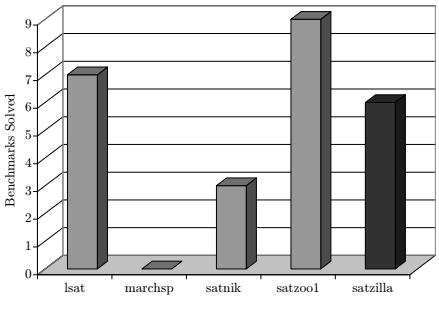

Fig. 15. SAT-2003 Handmade Category

Recently, we learned that `SATzilla` qualified to advance to the final round in the Random category. Solvers advancing in the other categories have not yet been named.

References

1. Fahiem Bacchus and Jonathan Winter. Effective preprocessing with hyper-resolution and equality reduction. In *SAT*, 2003.
2. B.Efron, T. Hastie, I. Johnstone, and R. Tibshirani. Regression shrinkage and selection via the lasso, 2002.
3. D. Le Berre and L. Simon. The essentials of the SAT 2003 competition. In *SAT*, 2003.
4. P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the Really Hard Problems Are. In *IJCAI-91*, 1991.
5. C. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. of Automated Reasoning*, 24(1), 2000.
6. Holger H. Hoos and Thomas Stutzle. Towards a characterisation of the behaviour of stochastic local search algorithms for SAT. *Artificial Intelligence*, 112(1-2):213–232, 1999.
7. Phokion Kolaitis. Constraint satisfaction, databases and logic. In *IJCAI*, 2003.
8. K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. Boosting as a metaphor for algorithm design. In *Constraint Programming*, 2003.
9. K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. A portfolio approach to algorithm selection. In *IJCAI-03*, 2003.
10. K. Leyton-Brown, E. Nudelman, and Y. Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *CP*, 2002.
11. L. Lobjois and M. Lemaître. Branch and bound algorithm selection by performance prediction. In *AAAI*, 1998.
12. R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity for characteristic 'phase transitions'. *Nature*, 400, 1998.
13. B. Selman, D. G. Mitchell, and H. J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1-2):17–29, 1996.
14. R. Tibshirani. Regression shrinkage and selection via the lasso, 1994.
15. D. Tompkins and H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In *SAT*, 2004.
16. R. Williams, C. Gomes, and B. Selman. Backdoors to typical case complexity. In *IJCAI*, 2003.