Published: 02 July 2009
Accepted: 01 March 2009
Revised: 01 October 2008
Received: 01 October 2005

Citation in BibTeX format

RESEARCH-ARTICLE

Empirical hardness models: Methodology and a case study on combinatorial auctions

KEVIN LEYTON-BROWN, The University of British Columbia, Vancouver, BC, Canada

EUGENE NUDELMAN, Stanford University, Stanford, CA, United States

Y SHOHAM, Stanford University, Stanford, CA, United States

Journal of the ACM (JACM), Volume 56, Issue 4 (June 2009)
https://doi.org/10.1145/1538902.1538906
EISSN: 1557-735X

Empirical Hardness Models: Methodology and a Case Study on Combinatorial Auctions

KEVIN LEYTON-BROWN

University of British Columbia, Vancouver, British Columbia, Canada

AND

EUGENE NUDELMAN AND YOAV SHOHAM

Stanford University, Stanford, California

Abstract. Is it possible to predict how long an algorithm will take to solve a previously-unseen instance of an NP-complete problem? If so, what uses can be found for models that make such predictions? This article provides answers to these questions and evaluates the answers experimentally.

We propose the use of supervised machine learning to build models that predict an algorithm's runtime given a problem instance. We discuss the construction of these models and describe techniques for interpreting them to gain understanding of the characteristics that cause instances to be hard or easy. We also present two applications of our models: building algorithm portfolios that outperform their constituent algorithms, and generating test distributions that emphasize hard problems.

We demonstrate the effectiveness of our techniques in a case study of the combinatorial auction winner determination problem. Our experimental results show that we can build very accurate models of an algorithm's running time, interpret our models, build an algorithm portfolio that strongly outperforms the best single algorithm, and tune a standard benchmark suite to generate much harder problem instances.

Categories and Subject Descriptors: I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Automatic analysis of algorithms*; *Program synthesis*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Graph and tree search strategies*; *Heuristic methods*; I.2.6 [**Artificial Intelligence**]: Learning

General Terms: Design, Economics, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Empirical analysis of algorithms, algorithm portfolios, combinatorial auctions, runtime prediction

This work was supported by DARPA grant F30602-00-2-0598, NSF grant IIS-0205633, the Intelligent Information Systems Institute, Cornell, an NSERC Discovery Grant and a Stanford Graduate Fellowship.

Authors' addresses: K. Leyton-Brown, Department of Computer Science, University of British Columbia, 2366 Main Hall, Vancouver, V6T 1Z4, British Columbia, Canada, e-mail: kevinlb@cs.ubc.ca; E. Nudelman and Y. Shoham, Department of Computer Science, Stanford University, Gates Bldg 1A-140, Stanford, CA 94305-9010, e-mail: {eugnud;shoham}@cs.stanford.edu.

DOI 10.1145/1538902.1538906 http://doi.acm.org/10.1145/1538902.1538906

ACM Reference Format:

Leyton-Brown, K., Nudelman, E., and Shoham, Y. 2009. Empirical hardness models: Methodology and a case study on combinatorial auctions. J. ACM 56, 4, Article 22 (June 2009), 52 pages. DOI = 10.1145/1538902.1538906 http://doi.acm.org/10.1145/1538902.1538906

1. *Introduction*

Many of the most interesting and important computational problems are NP-complete. This would seem to suggest that these problems are hopelessly intractable for all but the smallest instances. In practice, luckily, the situation is often much better, because NP-completeness is only a worst-case notion. In many domains of interest, instances that arise in practice are overwhelmingly easier than worst-case instances of the same size.

Recently, there has been substantial interest in understanding the "empirical hardness" (also called "typical-case complexity") of NP-complete problems. This line of work has focused on identifying factors that determine how hard distributions or individual instances of NP-complete problems will be for particular algorithms to solve in practice. Such work promises to advance both the practical development of state-of-the-art algorithms and also theoretical understanding.

1.1. Past Work: Empirical Hardness of Decision Problems. In recent years,[1] many researchers in the constraint programming and artificial intelligence communities have sought simple mathematical relationships between features of problem instances and the hardness of a problem. The majority of this work has focused on decision problems (i.e., problems that ask a yes/no question of the form, "Does there exist a solution meeting the given constraints?"). The most popular approach for understanding the empirical hardness of such problems—taken for example by Cheeseman et al. [1991] and Selman et al. [1996]—is to vary some parameter of the input looking for a easy–hard–less hard transition corresponding to a phase transition in the solvability of the problem. Some of the earliest work in this category concerned the satisfiability problem (SAT), which represents a generic constraint satisfaction problem with binary variables and arbitrary constraints. For example, Selman et al. [1996] considered the empirical performance of solvers running on uniformly randomly-generated SAT instances, and 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. Further, they demonstrated 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 an algorithm-independent property of the instance: the probability that a randomly-generated formula having a given ratio will be satisfiable. There has also been considerable work from the theory community bounding the point at which this phase transition occurs, particularly for uniform random 3-SAT and k-SAT, and describing how the easy–hard–less hard transition arises, particularly for simple backtracking-based (so-called DPLL) algorithms [Frieze and Suen 1996; Dubois and Boufkhad 1997; Beame et al. 1998; Dubois et al. 2000; Franco 2001; Achlioptas 2001; Cocco and Monasson 2004; Achlioptas et al. 2004].

[1] The work described in this article was concluded in 2005. Thus, our discussion of related literature concentrates on work done to that date, and does not attempt to exhaustively survey all newer work. Nevertheless, we have attempted to provide pointers to some of the most related recent work.

The last decade or so has seen increased enthusiasm for the idea of studying algorithm performance experimentally, using the same sorts of methods that are used to study natural phenomena. This work has complemented the theoretical worst-case analysis of algorithms, leading to interesting findings and concepts. For example, this approach was applied to the quasigroup completion problem [Gomes and Selman 1997]. Follow-up work took a closer look at runtime distributions [Gomes et al. 2000], demonstrating that runtimes of many SAT algorithms tend to follow power-law distributions and that random restarts provably improve such algorithms. Later, Gomes et al. [2004] refined these notions and models, demonstrating that statistical regimes of runtimes change drastically as one moves across the phase transition. A related approach to understanding empirical hardness rests on the notion of a backbone [Monasson et al. 1998; Achlioptas et al. 2000], which is the set of solution invariants. Williams et al. [2003] defined the concept of a backdoor of a CSP instance: the set of variables, which, if assigned correctly, lead to a residual problem that is solvable in polynomial time. They showed that many real world SAT instances indeed have small backdoors, which may explain the observed empirical behavior of SAT solvers. A lot of effort has also gone into the study of search space topologies for stochastic local search algorithms [Hoos and Stützle 1999; Hoos and Stützle 2004]. Finally, some work has been more theoretical in nature. For example, Kolaitis [2003] defined and studied "islands of tractability" of hard problems.

1.2. Past Work: Empirical Hardness of Optimization Problems.

Some researchers have also examined the empirical hardness of optimization problems, which ask a real-numbered question of the form, "What is the best solution meeting the given constraints?". Often, feasibility (the existence of a solution) is not a major hurdle in optimization problems; in such cases, a phase transition in solvability clearly cannot exist. One way of finding hardness transitions related to optimization problems is to transform them into decision problems of the form, "Does there exist a solution with objective function value $\geq x$?" This approach has yielded promising results, for example, when applied to MAX-SAT [Zhang 2001]; however, it is hard to apply when the expected value of the solution depends on input factors irrelevant to hardness (e.g., in MAX-SAT scaling of the weights has an effect on the objective function value, but not on the combinatorial structure of the problem). Other experimentally-oriented work includes the extension of the concept of backbone to optimization problems [Slaney and Walsh 2001], although it is often difficult to define for arbitrary problems and can be costly to compute.

A second approach is to attack the problem analytically rather than experimentally. For example, Zhang [1999] performed average-case analysis of particular classes of search algorithms. Though these results rely on independence assumptions about the branching factor and heuristic performance at each node of the search tree that do not generally hold, the approach made theoretical contributions—describing a polynomial/exponential-time transition in average-case complexity—and shed light on real-world problems. Korf and Reid [1998] predicted the average number of nodes expanded by a simple heuristic search algorithm such as A* on a particular problem class by making use of the distribution of heuristic values in the problem space. As above, strong assumptions were required: for example, that the branching factor is constant and node-independent, and that edge costs are uniform throughout the tree.

Both experimental and theoretical approaches have sets of problems to which they are not well suited. Existing experimental techniques have trouble when problems have high-dimensional parameter spaces, as it is impractical to manually explore the space of all relations between parameters in search of a phase transition or some other predictor of an instance's hardness. This trouble is compounded when many different data distributions exist for a problem, each with its own set of parameters. Similarly, theoretical approaches are difficult when the input distribution is complex or is otherwise hard to characterize. In addition, they tend to become intractable when applied to complex algorithms, or to problems with variable and interdependent edge costs and branching factors. Furthermore, they are generally unsuited to making predictions about the empirical hardness of individual problem instances, instead concentrating on average (or worst-case) performance on a class of instances.

1.3. Our Case Study Problem: Combinatorial Auction Winner Determination.

To evaluate our ideas on a real problem, in this article, we describe a case study of the combinatorial auction winner determination problem. Combinatorial auctions have received considerable attention from computer science and artificial intelligence researchers over the past several years because they provide a general framework for allocation and decision-making problems among self-interested agents. Specifically, agents may bid for bundles of goods, and are guaranteed that these bundles will be allocated "all-or-nothing". (For an introduction to the topic—as well as coverage of more advanced topics—see Cramton et al. [2006].) These auctions are particularly useful in cases where agents consider some goods to be *complementary*, which means that an agent's valuation for some bundle exceeds the sum of its valuations for the separate goods contained in the bundle. They also usually allow agents to specify that they consider some goods to be *substitutable*. This case is often addressed by allowing agents to state XOR constraints between bids, indicating that at most one of these bids may be satisfied.

The winner determination problem (WDP) is a combinatorial optimization problem that arises naturally in many combinatorial auctions. The issue is that it is not straightforward to determine which bids ought to win and which should lose. The goal of the WDP is to determine the feasible subset of the bids that maximizes the sum of the bid amounts. Formally, let $G = \{\gamma_1, \gamma_2, \ldots, \gamma_m\}$ be a set of goods, and let $B = \{b_1, \ldots, b_n\}$ be a set of bids. Bid b_i is a pair $(p(b_i), g(b_i))$ where $p(b_i) \in \mathbb{R}^+$ is the price offer of bid b_i and $g(b_i) \subseteq G$ is the set of goods requested by b_i. For each bid b_i define an indicator variable x_i that encodes the inclusion or exclusion of bid b_i from the allocation. We can now state the WDP formally.

Definition 1. The *Winner Determination Problem (WDP)* is:

$$\begin{aligned} \text{maximize:} \quad & \sum_i x_i \, p(b_i) & \\ \text{subject to:} \quad & \sum_{i \mid \gamma \in g(b_i)} x_i \le 1 & \forall \gamma \in G \\ & x_i \in \{0, 1\} & \forall i. \end{aligned}$$

The WDP is equivalent to the weighted set packing problem, and so is NP-complete. However, as we will discuss below, it has been observed that the WDP is often relatively easy to solve. Our goal in this article's case study is to investigate this phenomenon in more detail. The WDP is a good example of a problem that is ill-suited to study by either existing experimental or theoretical approaches to understanding empirical hardness. For example, WDP instances can be characterized by a large number of apparently relevant features. There exist many, highly parameterized instance distributions of interest to researchers. There is significant variation in edge costs throughout the search tree for most algorithms. Thus, if we are to study the empirical hardness of the WDP, none of the approaches surveyed above will suffice.

1.4. EMPIRICAL HARDNESS MODELS: OVERVIEW. Our main idea in this article is that machine learning techniques can be used to extract interesting information about an algorithm's empirical hardness automatically from huge bodies of experimental data. (Our work can be seen as part of a broader trend that leverages machine learning in more applied areas of computer science (e.g., Diao et al. [2003], Zheng et al. [2003], Goldszmidt [2007], and especially Goldsmith et al. [2007]). A particularly related early article is work by Brewer [1994, 1995] on using statistical models to automatically optimize high-level libraries for parallel processors; we describe it in detail in Section 6.2.1.1.) Our focus is on building what we call *empirical hardness models*—that is, models that can predict the amount of time a given algorithm will take to solve a given problem instance. In Section 2, we describe our methodology for constructing these models, and in Section 3 we apply this methodology to our WDP case study.

Before diving in, it is worthwhile to consider why we would want to be able to construct such models. First, sometimes it is simply useful to be able to predict how long an algorithm will take to solve a particular instance. This can allow the user to decide how to allocate computational resources to other tasks, whether the run should be aborted, and whether an approximate or incomplete (e.g., local search) algorithm should be used instead.

Second, it has often been observed that algorithms for NP-complete problems can vary by many orders of magnitude in their running times on different instances of the same size—even when these instances are drawn from the same distribution. (Indeed, we show that the WDP exhibits this sort of runtime variability in Figure 4.) However, little is known about what causes these instances to vary so substantially in their empirical hardness. In Section 4, we show how our run-time models can be analyzed to shed light on the sources of this variability, and in Section 5, we apply these ideas to our case study. This sort of analysis could lead to changes in problem formulations to reduce the chance of long solver runtimes. Also, better understanding of high run-time variance could serve as a starting point for improvements in algorithms that target specific problem domains.

Empirical hardness models also have other applications, which we discuss in Section 6. First, we show that accurate run-time models can be used to construct efficient algorithm portfolios by selecting the best among a set of algorithms based on the current problem instance. Second, we explain how our models can be applied to tune input distributions for hardness, thus facilitating the testing and development of new algorithms that complement the existing state of the art. Section 7 evaluates these ideas in our WDP case study.

We briefly survey our own previous work on empirical hardness models. We introduced the idea in Leyton-Brown, Nudelman and Shoham [2002] and studied its application to combinatorial winner determination. Later, we proposed the application of empirical hardness models to the construction of algorithm portfolios and hard instance distributions in a pair of companion papers [Leyton-Brown, Nudelman, Andrew, McFadden and Shoham, 2003a, 2003b]. We gave an overview of this work in a book chapter [Leyton-Brown, Nudelman and Shoham 2006]. The current article draws on and extends all this past work.

More recently, we have continued to study empirical hardness models in the context of the satisfiability problem. Nudelman, Leyton-Brown, Devkar, Shoham, and Hoos [2004b] first showed that empirical hardness models can be applied to SAT, and also proposed a portfolio-based SAT solver. We built such a solver, called SATzilla, and entered it into the international SAT competition in 2003, where it placed second in two categories and third in another (out of nine total) [Nudelman, Leyton-Brown, Devkar, Shoham and Hoos, 2004a]. Xu, Hutter, Hoos, and Leyton-Brown [2007b, 2008] subsequently improved SATzilla and entered it into the 2007 SAT competition, where it won in three categories, came second in another, and came third in still another. This shows (to our knowledge, for the first time) that algorithm portfolios are not just theoretical curiosities, but instead can achieve state-of-the-art performance in challenging, practical settings. Also in the context of SAT, Hutter, Hamadi, Hoos, and Leyton-Brown [2006] investigated the extension of empirical hardness models to randomized and incomplete algorithms and their application to automatic parameter tuning. Finally, Xu, Hoos, and Leyton-Brown [2007a] showed that machine learning can be used to accurately predict instances' satisfiability status and that these predictions in turn can be used in a mixture-of-experts framework to yield more accurate empirical hardness models. There are various other ways in which these more recent papers relate to the topics discussed in this article, we offer more details as we go along.

We have written software in Matlab for constructing empirical hardness models using the techniques we describe in this article, and for analyzing them and using them to build algorithm portfolios. The code and well as the data that we collected in our case study are available at `http://cs.ubc.ca/~kevinlb/downloads.html`.

2. *Building Empirical Hardness Models*

We propose the following methodology for predicting the running time of a given algorithm on individual instances of a problem such as WDP, where instances are drawn from some arbitrary distribution. As just mentioned this methodology was first introduced in earlier work [Leyton-Brown et al. 2002], though we present it somewhat differently here. We also note that it is related to approaches for statistical experiment design (see, e.g., Mason et al. [2003] and Chaloner and Verdinelli [1995]).

(1) *Select an algorithm of interest.* A black-box implementation is enough; no knowledge of the algorithm's internal workings is needed.

(2) *Select an instance distribution.* In practice, this may be achieved as a distribution over different instance generators, along with a distribution over each generator's parameters, including parameters that control problem size.

(3) *Identify a set of features.* These features, used to characterize problem instances, must be quickly computable and distribution independent. Eliminate redundant or uninformative features.

(4) *Collect data*. Generate a desired number of instances by sampling from the distribution chosen in Step (2). For each problem instance, determine the running time of the algorithm selected in Step (1), and compute all the features selected in Step (3). Split this data into a training set, a test set, and, if sufficient data is available, a validation set.

(5) *Learn a model*. Based on the training set constructed in Step (4), use a machine learning algorithm to learn a function mapping from the features to a prediction of the algorithm's running time. Evaluate the quality of this function on the test set.

In the rest of this section, we describe each of these points in detail.

2.1. STEP 1: SELECTING AN ALGORITHM. This step is simple: any algorithm can be chosen. Indeed, one advantage of our methodology is that it treats the algorithm as a black box, meaning that it is not necessary to have access to an algorithm's source code, etc. Note, however, that the empirical hardness model that is produced through the application of this methodology will be algorithm-specific, and thus can not reliably provide information about a problem domain transcending the particular algorithm or algorithms under study. (Sometimes, however, empirical hardness models may provide such information *indirectly*, when the observation that certain features are sufficient to explain hardness can serve as the starting point for theoretical work. Techniques for using our models to initiate such a process are discussed in Section 4. It is also possible that an empirical hardness model would transcend a particular algorithm if several algorithms give rise to similar hardness models: this would mean that the algorithms all have much in common, that the model is algorithm-agnostic, or a combination of both.) We do not consider the algorithm-specificity of our techniques to be a drawback—it is not clear whether algorithm-independent empirical hardness is even well defined—but the point nevertheless deserves emphasis.

Although this article discusses only deterministic algorithms, more recent work has also demonstrated that our methodology may be used to build empirical hardness models for both randomized tree search algorithms and stochastic local search algorithms [Nudelman et al. 2004b; Hutter et al. 2006; Xu et al. 2007a, 2008]. Based on the findings in that work, we observe that our techniques extend directly to cases where the algorithm's running time varies from one invocation to another. Indeed, even when we do restrict ourselves to deterministic algorithms, multiple instances (corresponding to different runtimes) will map to the same point in feature space. Incomplete algorithms are trickier, because the notion of running time is not always well defined when the algorithm can lack a termination condition. For example, on an optimization problem such as WDP an incomplete algorithm may never prove that it has found the optimal allocation. The situation is more straightforward when working with decision problems (e.g., SAT). In such domains, incomplete algorithms will terminate when they find a solution. It is true that a local search algorithm will never terminate when given an unsatisfiable instance. However, in practice it is not uncommon for even deterministic algorithms to fail to solve a substantial fraction of the instances within the available time; we can handle the case of a local search algorithm that does not terminate using the same techniques. As discussed in the introduction, optimization problems can be converted to decision problems by asking whether a solution exists with an objective function value of at least some amount k. Leveraging this idea, models for incomplete optimization algorithms can predict the amount of time the algorithm will take to exceed an objective function value of k.

Another way of working with incomplete algorithms is to predict solution quality directly rather than predicting the amount of time the algorithm will run. While we have also had some (as yet unpublished) success with the latter in the Traveling Salesman Problem domain, in this article we focus exclusively on running time as it is the most natural and universal measure.

2.2. STEP 2: SELECTING AN INSTANCE DISTRIBUTION. Any instance distribution can be used to build an empirical hardness model. In the experimental results presented in this article we consider instances that were created by artificial instance generators; however, real-world instances may also be used (see, e.g., Nudelman et al. [2004a] and Xu et al. [2008]). The key point that we emphasize in this step is that instances should always be understood as coming from some distribution or as being generated from some underlying real-world problem. The learned empirical hardness model will only describe the algorithm's performance on this distribution of instances—while a model may *happen* to generalize to other problem distributions, there is no guarantee that it will do so. Thus, the choice of instance distribution is critical. Of course, this is the same issue that arises in any empirical work: whenever an algorithm's performance is reported on some data distribution, the result is only interesting insofar as the distribution is important or realistic, and cannot be relied upon to generalize.

It is often the case that in the literature on a particular computational problem, a wide variety of qualitatively different instance distributions have been proposed. Sometimes one's motivation for deciding to build empirical hardness models will be tied to a very particular domain, and the choice of instance distribution will be clear. In the absence of a reason to prefer one distribution over another, we favor an approach in which a new, hybrid distribution is constructed, in which one of the original distributions is chosen at random and then an instance is drawn from the chosen distribution. In a similar way, individual instance generators often have many parameters; rather than fixing parameter values, we prefer to establish a range of reasonable values for each parameter and then to generate each new instance based on parameters drawn at random from these ranges.

One class of distribution parameters deserves special mention: parameters that control problem size. These are special because they describe a source of empirical hardness in NP-complete problem instances that is already at least partially understood. In particular, as problems get larger they usually also get harder to solve. However, as we illustrate in our case study (Section 3.2.3), there can be multiple ways of defining problem size for a given problem. One situation in which defining problem size is important is when the reason for building an empirical hardness model is understanding what *other* features of instances are predictive of hardness. In this case, we define a distribution in which problem size is held constant, allowing our models to use other features to explain remaining variation in runtime. In other cases, we may want to study a distribution in which problem size varies; even so, it is important to ensure that problem size is clearly defined so that this variation may be understood correctly. Another advantage of having problem size defined explicitly is that its relationship to hardness may be known, at least approximately. Thus, it might be possible to tailor the hypothesis space used by the machine learning algorithm (Step 5) to make direct use of this information.

2.3. Step 3: Selecting Features. An empirical hardness model is a mapping from a set of features that describe a problem instance to a real value representing the modeled algorithm's predicted runtime. Clearly, choosing good features is crucial to the construction of good models. Unfortunately there is no known, automatic way of constructing good feature sets. Instead, researchers must use domain knowledge to identify properties of the instance that appear likely to provide useful information. However, we did discover that many intuitions can be generalized. For example, features that proved useful for one constraint satisfaction or optimization problem can be useful to other problems as well. Also, constraint relaxations or simplified algorithms can often give rise to good features.

The good news is that techniques *do* exist for building good models even if the set of features provided includes many redundant or useless features. These techniques are of two kinds: one approach throws away useless or harmful features, while the second keeps all of the features but builds models in a way that uses features only to the extent that they are helpful. Because of the availability of these techniques, we recommend that researchers brainstorm a large list of features that have the possibility to prove useful and then allow models to select among them.

After a set of features has been identified, features that are extremely highly correlated with other features or are extremely uninformative (e.g., always take the same value) should be eliminated immediately, on the basis of some small initial experiments. Features that are not (almost) perfectly correlated with other features should be preserved at this stage, but should be re-examined if problems occur in Step (5) (e.g., numerical problems arise in the training of models; models do not generalize well).

We restrict the sorts of features available to empirical hardness models in two ways. First, we only consider features that can be generated from *any* problem instance, without knowledge of how that instance was constructed. For example, we consider it appropriate to use graph-theoretic properties of the constraint graph (e.g., node degree statistics) but not parameters of a specific distribution used to generate an instance. Second, we restrict ourselves to those features that are computable in low-order polynomial time, since the computation of the features should scale well as compared to solving the optimization problem.

2.4. Step 4: Collecting Data. This step is simple to explain, but nontrivial to actually perform. In the case studies that we have performed, we have found the collection of data to be very time-consuming both for our computer cluster and for ourselves.

First, we caution that it is important not to attempt to build empirical hardness models with an insufficient body of data. Since each feature that is introduced in Step 3 increases the dimensionality of the learning problem, a very large amount of data may be required for the construction of good models.[2] Fortunately, problem instances are often available in large quantities, and so the size of a data set is

[2] In fact, the effective dimensionality of the learning problem depends on the degree of correlation between the features. Since, in practice, features tend to be very highly correlated, substantially less data may be required than would be needed if features were entirely uncorrelated. Indeed, in unpublished experiments, we found that results qualitatively similar to those we report in this article can be obtained with dramatically smaller data sets.

often limited only by the amount of time for which one is willing to wait. This tends to encourage the use of large parallel computer clusters, which—luckily—are increasingly available. In this case, it is essential to ensure that hardware is identical throughout the cluster, that job migration is disallowed, and that no node runs more jobs than it has processors, in order to ensure that time measurements are reliable and comparable.

Second, when one's research goal is to characterize an algorithm's empirical performance on hard problems, it is important to run problems at a size for which preprocessors do not have an overwhelming effect, and at which the runtime variation between hard and easy instances is substantial. Thus, while easy instances may take a small fraction of a second to solve, hard instances should take many hours. (We see this sort of behavior in our WDP case study, for example in Section 3.4.) Since the run-time distribution may be heavy tailed, it can be infeasible to wait for every run to complete. In this case, it is necessary to cap runs at some maximum amount of time.[3] In our experience, such capping is reasonably safe as long as the captime is chosen in a way that ensures that only a small fraction of the instances will be capped. All the same, capping should always be performed cautiously.

Finally, we have found data collection to be logistically challenging. When experiments involve tens or hundreds of processors and many CPU-years of computation, jobs will crash, data will get lost, research assistants will graduate, and bugs will be discovered in feature-computation code. In the work that led to this article, we have learned a few general lessons. (None of these observations is especially surprising—in essence, they all boil down to a recommendation to invest time in setting up clean data collection methods rather than adopting quick and dirty approaches.) First, enterprise-strength queuing software should be used rather than attempting to dispatch jobs using homemade scripts. Second, data should not be aggregated by hand, as portions of experiments will sometimes need to be rerun and such approaches easily become unwieldy. Third, for the same reason the instances used to generate data should always be kept. When instances are prohibitively large, it can sometimes suffice to store generator parameters along with the random seed used to generate the instance. In the same vein, it pays to use some source control system to keep track of all scripts and programs used to generate data. Finally, it is worth the extra effort to store experimental results in a database rather than writing output to files—this reduces headaches arising from concurrency, and also makes queries much easier.

2.5. STEP 5: BUILDING MODELS. Our methodology is agnostic about what particular machine learning algorithm should be used to construct empirical hardness models. Since the goal is to predict runtime, which is a real-valued variable, we have come to favor the use of statistical regression techniques as our machine learning tool. In our initial (unpublished) work we considered the use of classification approaches such as decision trees, but ultimately we became convinced that these

[3] In the first data sets of our WDP case study we capped runs at a maximum number of *nodes*; however, we now believe that it is better to cap runs at a maximum running time, which we did in our most recent WDP data set. The reason that we prefer time-based capping is that per-node computation can vary substantially within and across instances. We revisit the issue of capping runtimes in Section 6.2.2, and discuss some more recent ideas on the subject in Footnote 21.

approaches were less appropriate. (For a discussion of some of the reasons that we drew this conclusion, see Section 6.2.1.3.) Hutter et al. [2006] investigated nonparametric methods such as Gaussian processes; they yield better models in some cases and have the advantage of offering uncertainty estimates. In the end we have tended away from these techniques because the gains in predictive accuracy that they offer tend to be small, because of their computational complexity, and because they tend to produce models whose size grows with the amount of data used in training.[4] Relatedly, although we focus on frequentist regression techniques here, there are cases where it is appropriate to consider the use of Bayesian alternatives. Hutter et al. [2006] made use of sequential Bayesian linear regression; this yields the same runtime predictions as (frequentist) linear regression, but offers the advantage of uncertainty estimates within a parametric framework. Finally, it is possible to make use of mixtures-of-experts approaches, selecting the experts based on a prediction of the objective function value [Xu et al. 2007a] and/or based on a prediction of the underlying distribution that generated the instance [Xu et al. 2008].

There are a wide variety of different parametric, frequentist regression techniques. The most appropriate for our purposes perform supervised learning. A large literature addresses these statistical techniques; for an introduction, see, for example, Hastie et al. [2001]. Such techniques choose a function from a given hypothesis space (i.e., a set of candidate mappings from the 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 select good mappings from this hypothesis space and can be tractably minimized.

The simplest supervised regression technique is linear regression, which learns functions of the form $\sum_i w_i f_i$, where f_i is the ith feature and the w's are free variables, and which has as its error metric root mean squared error (RMSE). Linear regression is a computationally appealing procedure because it reduces to the (roughly) cubic-time problem of matrix inversion.[5] In comparison, most other regression techniques depend on more complex optimization problems such as quadratic programming. Besides being relatively tractable and well understood, linear regression has another advantage that is very important for this work: it produces models that can be analyzed and interpreted in a relatively intuitive way, as we show in Section 4. While we discuss other regression techniques later in this article (e.g., support vector machine regression, ridge regression, lasso regression, multivariate adaptive regression splines), we present linear regression as our baseline machine learning technique.

Linear regression is implemented as follows. Let D be a matrix where each row corresponds to an individual runs of the algorithm on a different problem instance

[4] For this last reason, we do not discuss other nonparametric methods such as nearest neighbor or random forests of regression trees in this article, though in some recent unpublished work we have had some success with the latter method.

[5] In fact, the worst-case complexity of matrix inversion is $O(N^{log_2 7}) = O(N^{2.808})$.

and each column corresponds to a different feature. Thus, cells in the matrix denote the value of a given feature on a given instance. Furthermore let the matrix contain one final feature that is always one; this permits the model to have a non-zero intercept (i.e., to predict a non-zero run-time even when all feature values are zero). Let r denote a column vector whose elements denote the algorithm's observed run-time on each problem instance. We can then train a linear model w that maps from observed feature values to a predicted run-time as

$$w = (D^T D)^{-1} D^T r.$$

Observe that w is the vector of weights described above, plus the weight for the constant feature. Note that the expression above requires that the matrix $D^T D$ be nonsingular.

2.5.1. *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. This is the right error metric to use when data labels (runtimes) are generated by a linear function and then subjected to Gaussian noise having unvarying standard deviation. Because this measure penalizes outlying points superlinearly, it can be inappropriate in cases where the data contains many outliers. Some regression techniques use ℓ_1 error, which penalizes outliers linearly; however, optimizing these error metrics often relies on quadratic programming.

Some error metrics express an additional preference for models with small (or even zero) coefficients to models with large coefficients. This can lead to more reliable models on test data, particularly when 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, which is tuned using either cross-validation or a validation set.

In our own work, we have tended to use ridge regression, in part because it can provide increased numerical stability for the matrix inversion. Ridge regression can be implemented as a simple extension of linear regression,

$$w = (D^T D + \delta I)^{-1} D^T r.$$

I denotes an identity matrix with number of rows and columns equal to the number of features plus one, and δ is the shrinkage parameter. Observe that with $\delta = 0$ we recover linear regression. In practice, we used very small values of δ simply to provide numerical stability. Experiments using the validation set or cross-validation with the training set can be used to determine whether larger values of δ also lead to improved prediction accuracy.

2.5.2. *Choosing a Hypothesis Space.* Although linear regression may seem quite limited, it can be extended to a wide range of nonlinear hypothesis spaces. There are two key tricks, both of which are quite standard in the machine learning literature. The first is to introduce new features that are functions (so-called "basis function expansions") of the original features. For example, in order to learn a model that is a second-order 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. Of course, we can use the same idea to reduce many other nonlinear hypothesis spaces to linear regression: all hypothesis spaces that can be expressed by $\sum_i w_i g_i(\mathbf{f})$, where the g_i's are arbitrary basis functions and $\mathbf{f} = \{f_i\}$. The key problem with such an approach is that the size of the new set of features is potentially much larger than the size of the original feature set, which may cause the regression problem to become intractable (e.g., preventing the feature matrix from fitting into memory). This problem can be mitigated through the use of kernel functions; however, that can make it difficult to use the subset selection approaches we discuss in Section 4. There is also a more general problem: using a more expressive hypothesis space can lead to overfitting, because the model can become expressive enough to fit noise in the training data. One way of dealing with this problem is to employ one of the shrinkage techniques discussed in Section 2.5.1. Alternately, it can make sense to add only a subset of the pairwise products of features; for example, only pairwise products of the k most important features in the linear regression model. Finally, a last problem is that if many of the new features are highly correlated with each other, numerical stability can be undermined.

In practice, we have found the problem of numerical instability to be significant when performing a basis function expansion, especially when the original features already exhibit strong correlation. In order to improve numerical stability and hence model accuracy, the following preprocessing technique may be used to weed out useless features:

(1) For each feature construct a regression model where that feature is used as the response variable, and all other features are used to predict it.

(2) Measure the adjusted R^2 of each such model to assess its prediction quality.[6]

(3) Identify the set of features whose models have adjusted R^2 above some threshold τ. (Observe that the closer any model's adjusted R^2 value gets to 1, the more redundant is the feature being predicted by that model.)

(4) If the set identified in Step (3) is nonempty, drop the feature whose model had the highest adjusted R^2, then repeat Steps (1)–(3). Otherwise, stop.

Sometimes we may want to consider hypothesis spaces of the form $h(\sum_i w_i g_i(\mathbf{f}))$. 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 $h(\sum_i w_i g_i(\mathbf{f}))$. 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.

In this article, unless otherwise noted, we use exponential models ($h(y) = 10^y$; $h^{-1}(y) = \log_{10}(y)$). In our experiments, we have generally found exponential models to be more suitable than linear models. This suggests that the run-time distributions that we are attempting to estimate are exponential, or at least are better modeled as exponential than as linear.

[6] Adjusted R^2 measures the percentage of variation in the response variable that can be explained using a linear combination of the features; we discuss it in Section 3.5.1.

3. *Building Empirical Hardness Models: WDP Case Study*

In this section, we consider again the steps of the methodology described in Section 2, and explain how we performed each step in our case study of the WDP.

3.1. STEP 1: SELECTING AN ALGORITHM. There are many WDP algorithms from which to choose, as much recent work has addressed the design of such algorithms. A very influential early article was Rothkopf et al. [1998], but it focused on tractable subcases of the problem and addressed computational approaches to the general WDP only briefly. The first algorithms designed specifically for the general WDP were proposed by Sandholm [1999] and Fujishima et al. [1999]; these algorithms were subsequently improved and extended upon in Sandholm et al. [2001] and Leyton-Brown et al. [2000a]. BoB [Sandholm 1999] and CASS [Fujishima et al. 1999] made use of classical AI heuristic search techniques, structuring their search by branching on bids and goods respectively.

More recently, there has been an increasing interest in solving the WDP with branch-and-bound search algorithms, using a linear-programming (LP) relaxation of the problem as a heuristic. ILOG's CPLEX software has come into wide use, particularly after influential arguments by Nisan [2000] and Anderson et al. [2000] and since the mixed integer programming module in that package improved substantially in version 6 (released 2000), and again in version 7 (released 2001). Starting with version 7.1, this off-the-shelf software reached the point where it was competitive with the best special purpose software, Sandholm et al.'s CABOB [Sandholm et al. 2001, 2005]. (In fact, CABOB makes use of CPLEX's linear programming package as a subroutine and also uses branch-and-bound search.) Likewise, GL [Gonen and Lehmann 2001] is a branch-and-bound algorithm that uses CPLEX's LP solver as its heuristic. Thus, the combinatorial auctions research community has seen convergence towards branch-and-bound search with an LP heuristic as a preferred approach for optimally solving the WDP. We chose to select CPLEX as our algorithm for this part of the case study, since it is at least comparable in performance to CABOB.[7]

3.2. STEP 2: SELECTING AN INSTANCE DISTRIBUTION. The wealth of research into algorithms for solving the WDP created a need for instances on which to test these algorithms. To date relatively few unrestricted combinatorial auctions have been held, or at least little data has been publicly released from whatever auctions *have* been held. Thus, researchers have tended to evaluate their WDP algorithms using artificial distributions.

3.2.1. *Legacy Data Distributions.* Along with the first wave of algorithms for the WDP, seven distributions were proposed [Sandholm 1999; Fujishima et al. 1999; Hoos and Boutilier 2000]. These distributions have been widely used by other researchers including many of those cited above. Each of these distributions may be seen as an answer to two questions: what number of goods to request in a bundle, and what price to offer for a bundle. Given a required *number* of goods, all

[7] We should mention that the CABOB algorithm continues to be developed commercially through CombineNet, Inc. (`http://www.combinenet.com`). Our discussion of CABOB is based on publicly released information about the algorithm [Sandholm et al. 2001, 2005] and may not apply to more recent versions. We wanted to include CABOB in our tests, but were not able to acquire a copy.

distributions select *which* goods to include in a bid uniformly at random without replacement. We give here the names that we used to identify them as "legacy" distributions in Leyton-Brown et al. [2000b].

—**L1**, the *Random* distribution from Sandholm [1999], chooses a number of goods uniformly at random from $[1, m]$, and assigns the bid a price drawn uniformly from [0, 1].

—**L2**, the *Weighted Random* distribution from Sandholm [1999], chooses a number of goods g uniformly at random from $[1, m]$ and assigns a price drawn uniformly from $[0, g]$.

—**L3**, the *Uniform* distribution from Sandholm [1999], sets the number of goods to some constant c and draws the price offer from [0, 1].

—**L4**, the *Decay* distribution from Sandholm [1999] starts with a bundle size of 1, and increments the bundle size until a uniform random draw from [0, 1] exceeds a parameter α.

—**L5**, the *Normal* distribution from Hoos and Boutilier [2000], draws both the number of goods and the price offer from normal distributions.

—**L6**, the *Exponential* distribution from Fujishima et al. [1999], requests g goods with probability $Ce^{-g/q}$, and assigns a price offer drawn uniformly at random from $[0.5g, 1.5g]$.

—**L7**, the *Binomial* distribution from Fujishima et al. [1999], gives each good an independent probability of p of being included in a bundle, and assigns a price offer drawn uniformly at random from $[0.5g, 1.5g]$ where g was the number of goods selected.

3.2.2. *CATS Distributions.* The above distributions have been criticized in several ways, perhaps most significantly for lacking economic justification [see, e.g., Anderson et al. 2000; Leyton-Brown et al. 2000b; de Vries and Vohra 2003]. This criticism was significant because the WDP is ultimately a weighted set packing problem; if the data on which algorithms are evaluated lacks any connection to the combinatorial auction domain, it is reasonable to ask what connection the algorithms have to the WDP in particular. To focus algorithm development more concretely on combinatorial auctions, in our past work [Leyton-Brown et al. 2000b] we introduced a set of benchmark distributions called the Combinatorial Auction Test Suite (CATS). By modeling bidders explicitly and determining bid amounts, sets of goods and sets of substitutable bids based on models of bidder valuations and models of problem domains, the CATS distributions aimed to serve as a step towards a realistic set of test distributions. For example, as mentioned earlier, none of the "legacy" distributions introduce any structure in the choice of *which* goods are included in a bundle. All of the CATS distributions do exhibit such structure.

The Combinatorial Auction Test Suite consists of five distributions: *paths*, *regions*, *arbitrary*, *matching*, and *scheduling*. We provide a high-level description of each distribution below. A detailed description of each distribution is given in Leyton-Brown et al. [2000b].

—**Paths** models an auction of transportation links between cities, or more generally of edges in a nearly-planar graph. Bids request sets of goods that correspond to paths between randomly selected pairs of nodes; substitutable bids are those that connect the same pairs of nodes; prices depend on path length.

—**Regions** models an auction of real estate, or more generally of any goods over which two-dimensional adjacency is the basis of complementarity; bids request goods that are adjacent in a planar graph.

—**Arbitrary** is similar to regions, but relaxes the planarity assumption and models arbitrary complementarities between discrete goods such as electronics parts or collectibles.

—**Matching** models airline take-off and landing rights auctions such as those that have been discussed by the FAA; each bid requests one take-off and landing slot bundle, and each bidder submits an XOR'ed set of bids for acceptable bundles.

—**Scheduling** models a distributed job-shop scheduling domain, with bidders requesting an XOR'ed set of resource time-slots that will satisfy their specific deadlines.

In Leyton-Brown et al. [2000b], we did not tune these distributions towards generating hard instances. Based on experimental evidence, some researchers have remarked that some CATS problems are comparatively easy in practice (see, e.g., Gonen and Lehmann [2000] and Sandholm et al. [2001]). In Section 3.4, we show experimentally that some CATS distributions are always very easy for CPLEX, while others can be extremely hard. We propose techniques for making these distributions computationally harder in Section 7.2.

3.2.3. *Defining Problem Size.* For the WDP, it is well known that problems become harder as the number of goods and bids increases.[8] For this reason, researchers have traditionally reported the performance of their WDP algorithms in terms of the number of bids and goods of the input instances. While it is easy to fix the number of goods, holding the number of bids constant is not as straightforward as it might appear.

Most special-purpose algorithms make use of a polynomial-time preprocessing step that removes bids that are dominated by one other bid.

Definition 2. Bid b_i is *weakly dominated by* bid b_j if $g(b_j) \subseteq g(b_i)$ and $p(b_i) \leq p(b_j)$.

If a distribution gives rise to a large number of dominated bids, it is possible that the apparent size of the problems given as input to WDP algorithms could be much greater than the size of the problem that leaves the preprocessor. This means that we might be somewhat misled about the core algorithms' scaling behaviors—whatever we observe will reflect on the preprocessors' scaling behavior as well.[9]

[8] An exception is that problems generally become easier when the number of bids grows *very* large in distributions favoring small bundles, because each small bundle is sampled much more often than each large bundle, giving rise to a new distribution for which the optimal allocation tends to involve only small bundles. To our knowledge, this was first pointed out by Anderson et al. [2000].

[9] Of course, many other polynomial-time preprocessing steps are possible, for example, a check for bids that are dominated by a pair of other bids. Indeed, CPLEX employs many, much more complex preprocessing steps before initiating its own branch-and-bound search. Our own experience with algorithms for the WDP has suggested that as compared to the removal of dominated bids, other polynomial-time preprocessing steps offer much poorer performance in terms of the number of bids discarded in a given amount of time. In any case, the results we give in this section suggest that domination checking should not be disregarded, since distributions differ substantially in the ratio between the number of nondominated bids and the raw number of bids.

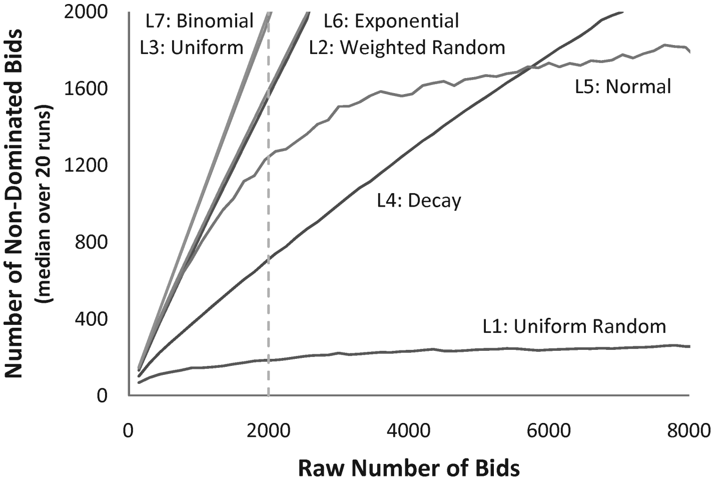

FIG. 1. Non dominated bids vs. raw bids.

Of course, it is not clear how much stock we should place in abstract arguments like those above—it is not clear whether the removal of dominated bids should be expected to have a substantial impact on algorithm behavior, or whether the relationship between the average number of nondominated bids and total bids should be expected to vary substantially from one distribution to another. To gain a better understanding, we set out to measure the relationship between numbers of dominated and undominated bids generated for all of our distributions.

Overall, we found that the CATS distributions generated virtually no dominated bids.[10] On the other hand, the legacy distributions were much more variable. Figure 1 shows the number of nondominated bids generated as a function of the total number of bids generated for each of the seven legacy distributions. In these experiments, each data series represents an average over 20 runs. Bids were generated for an auction having 64 goods, and we terminated bid generation once 2000 nondominated bids had been created.

3.2.4. *Distributions Used in Our Case Study.* Based on these results, we elected not to follow the traditional practice of defining problem size as the pair (*number of goods, number of bids*), and defined it instead as (*number of goods, number of nondominated bids*). We used all of the generators that were able to generate problems of any requested size. This led us to rule out the distributions L1 and L5: they often failed to generate a target number of nondominated bids even after millions of bids had been created.[11] We therefore elected to randomize uniformly over the 10 generators L2, L3, L4, L6, L7, *paths*, *regions*, *arbitrary*, *matching* and *scheduling*.

[10] This occurred largely because most bids generated included a bidder-specific "dummy good," preventing them from being dominated by bids from other bidders.

[11] This helps to explain why researchers have found that their algorithms perform well on L1 (see, e.g., Sandholm [1999] and Fujishima et al. [1999]) and L5 (see, e.g., Hoos and Boutilier [2000]).

Bid-Good Graph Features:

1–4. **Bid node degree statistics**: average, maximum, minimum and standard deviation of the bid nodes' degrees.

5–8. **Good nodes degree statistics**: average, maximum, minimum and standard deviation of the good nodes' degrees.

Bid Graph Features:

(9) **Edge Density**: number of edges in the BG divided by the number of edges in a complete graph having the same number of nodes.

10–15. **Node degree statistics**: the max and min, standard deviation, first and third quartiles, and median of nodes' degrees in the BG.

16–17. **Clustering Coefficient and Deviation**. A measure of "local cliquiness." For each node calculate the number of edges among its neighbors divided by $k(k-1)/2$, where k is the number of neighbors. We record average (the clustering coefficient) and standard deviation.

(18) **Average minimum path length**: the average minimum path length, over all pairs of bids.

(19) **Ratio of the clustering coefficient to the average minimum path length**: a measure of the smallness of the BG.

20–22. **Node eccentricity statistics**: The eccentricity of a node is the length of a shortest path to a node furthest from it. We calculate the minimum eccentricity of BG (graph radius), average eccentricity, and standard deviation of eccentricity.

LP-Based Features:

23–25. ℓ_1, ℓ_2, ℓ_∞ norms of the integer slack vector.

Price-Based Features:

(26) **Standard deviation of prices among all bids**: $\text{stdev}(p(b_i))$.

(27) **Deviation of price per number of goods**: $\text{stdev}(p(b_i)/|g(b_i)|)$.

(28) **Deviation of price per square root of the number of goods**: $\text{stdev}(p(b_i)/\sqrt{|g(b_i)|})$.

Problem Size Features:

(29) Number of goods.

(30) Number of non-dominated bids.

FIG. 2. Five groups of features. Only the first four were used in our fixed-size data sets.

Since the purpose of our case study was both to demonstrate that it is possible to construct accurate and useful runtime models and to home in on unknown sources of hardness, we ran experiments on both fixed and variable-sized data. Specifically, we constructed three fixed-size distributions, in which we set the problem size to (256 goods, 1000 nondominated bids), (144 goods, 1000 nondominated bids), and (64 goods, 2000 nondominated bids). Our variable-sized distribution drew the number of goods uniformly from [40, 400] and the number of nondominated bids uniformly from [50, 2000]. The collection of data from these distributions is discussed in more detail in Section 3.4.

3.3. STEP 3: SELECTING FEATURES.

We determined 37 features that we thought could be relevant to the empirical hardness of WDP, ranging in their computational complexity from linear to cubic time. We generated these feature values for all our problem instances, and then examined our data to identify and eliminate features that were always redundant. We were left with 30 features, which are summarized in Figure 2. We describe our features in more detail below, and also mention some of the redundant features that we eliminated.

There are two natural graphs associated with each instance; schematic examples of these graphs appear in Figure 3. First is the *bid-good graph* (BGG): a bipartite graph having a node for each bid, a node for each good and an edge between a bid and a good node for each good in the given bid. We measure a variety of BGG's properties: average, maximum, minimum, and standard deviation of the degrees of each node type.

The *bid graph* (BG) has an edge between each pair of bids that cannot appear together in the same allocation. This graph can be thought of as a constraint graph for the associated constraint satisfaction problem (CSP). As is often the case with the constraint graphs of CSPs, the BG captures a lot of useful information about the problem instance. Our second group of features are concerned with structural properties of the BG. We considered using the number of connected components of the BG to measure whether the problem was decomposable

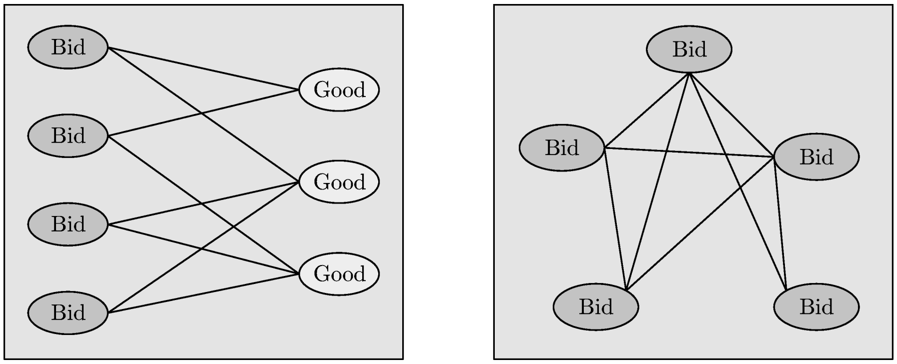

FIG. 3. Examples of the graphs used in calculating features 1–22: bid-good graph (left); bid graph (right).

into simpler instances, but found that virtually every instance consisted of a single component.[12]

The third group of features is calculated from the solution vector of the linear programming (LP) relaxation of the WDP. Recall that WDP can be formulated as an integer program (Definition 1). To obtain the LP relaxation, we simply drop the integrality constraints. We calculate the *integer slack* vector by replacing each component x_i with $|0.5 - x_i|$, measuring each component's distance from integrality. We converted these vectors into scalar features by taking ℓ_1, ℓ_2 or ℓ_∞ norms. These features appeared promising both because the slack gives insight into the quality of CPLEX's initial solution and because CPLEX uses LP as its search heuristic. Originally we also included median integer slack, but excluded the feature when we found that it was always zero on our data.

Our fourth group of features is the only one that explicitly considers the prices associated with bids. Observe that the scale of the prices has no effect on hardness; however, the spread is crucial, since it impacts pruning. We note that feature 28 was shown to be an optimal bid-ordering heuristic for certain greedy WDP approximation schemes by Gonen and Lehmann [2000].

Finally, we have the two features that define an instance's problem size. It is unremarkable that we included these features for our variable-size data set. More surprisingly, we also kept these features for our fixed-size data. The reason is that due to the way the CATS generators work, the actual number of goods and bids can sometimes be slightly different from the number requested. Thus, we did observe some variation in the numbers of goods and bids even in the fixed-size case. (In contrast, the legacy generators always produce the requested numbers of bids and goods.)

[12] It would also have been desirable to include some measure of the size of the (unpruned) search space. For some problems branching factor and search depth are used; for WDP neither is easily estimated. A related measure is the number of maximal independent sets of BG, which corresponds to the number of feasible solutions. However, this counting problem is hard, and to our knowledge does not have a polynomial-time approximation. Monte Carlo sampling approaches may be useful here.

3.4. STEP 4: COLLECTING DATA. As discussed in Section 3.2.4, our case study examined three fixed-size distributions and one variable-size distribution. The former were studied in our previously published conference papers on empirical hardness models [Leyton-Brown et al. 2002, 2003b, 2003a]. These data sets were collected using CPLEX 7.1. Our variable-size data set is new, and represents roughly three times as much computer time. Because a new version of CPLEX became available before we began to construct the variable-size data set, we upgraded our CPLEX software at this point and collected the variable-sized data set using CPLEX 8.0. We now describe these data sets in more detail.

For our fixed-size experiments we generated three separate data sets at different problem sizes, to ensure that our results were not artifacts of one particular choice of problem size. In our first data set, we ran CPLEX on WDP instances from each of our 10 distributions at a problem size of 1000 nondominated bids and 256 goods each. We collected a total of 4987 instances, or roughly 500 instances per distribution. The second data set was at a problem size of 1000 nondominated bids and 144 goods with a total of 4257 instances; the third data set was at a problem size of 2000 bids and 64 goods, and we ran CPLEX on 4428 instances.[13] Where we present results for only one fixed-size data set, the first (1000/256) data set is always used. All of our fixed-size data was collected by running CPLEX version 7.1 with preprocessing turned off. We used a cluster of 4 machines, each of which had 8 Pentium III Xeon 550 MHz processors and 4G RAM and was running Linux 2.2.12. Since many of the instances turned out to be exceptionally hard, we stopped CPLEX after it had expanded 130,000 nodes (reaching this point took between 2 hours and 22 hours, averaging 9 hours). Overall, solution times varied from as little as 0.01 seconds to as much as 22 hours. We spent approximately 3 years of CPU time collecting this data. We also computed our 30 features for each instance. (Recall that feature selection took place after all instances had been generated.)

Our variable-size data set was collected several years later. The number of requested bids was randomly selected from the interval [50, 2000], and the number of requested goods was drawn from [40, 400]. We obtained runtimes for CPLEX 8.0 using default parameters on 8371 of these instances (roughly 800 instances per distribution). This time we capped CPLEX based on runtime rather than the number of nodes. The timeout for CPLEX was set to 1 CPU-week (i.e., 604,800 seconds of CPU time). This timeout was reached only on two instances from the CATS Arbitrary distribution. The variable-size data set was collected on a cluster of 12 dual-CPU 2.4GHz Xeon machines running Linux 2.4.20. The average run of CPLEX took around 4 hours on this data set, and the whole data set took almost 9 CPU-years to collect![14]

For all of our data sets, we randomly divided our data into a training set, a validation set, and a test set, according to the ratio 70:15:15. We used the training set

[13] We experienced some initial problems with the CATS Paths distribution (now fixed). As a result, we do not have Paths data in all of our data sets: we managed to rerun experiments to include this distribution for the 1000 bid/256 good data set, but we dropped Paths from the other data sets. We observed that the performance of our models was qualitatively the same whether Paths was included or excluded.

[14] Especially attentive readers will notice that the numbers do not work out—8371 instances at an average of 4 hours does not add up to 9 years. What's missing is that we also ran the CASS algorithm on the same instances. This part of our data collection effort is described in Section 7.1.

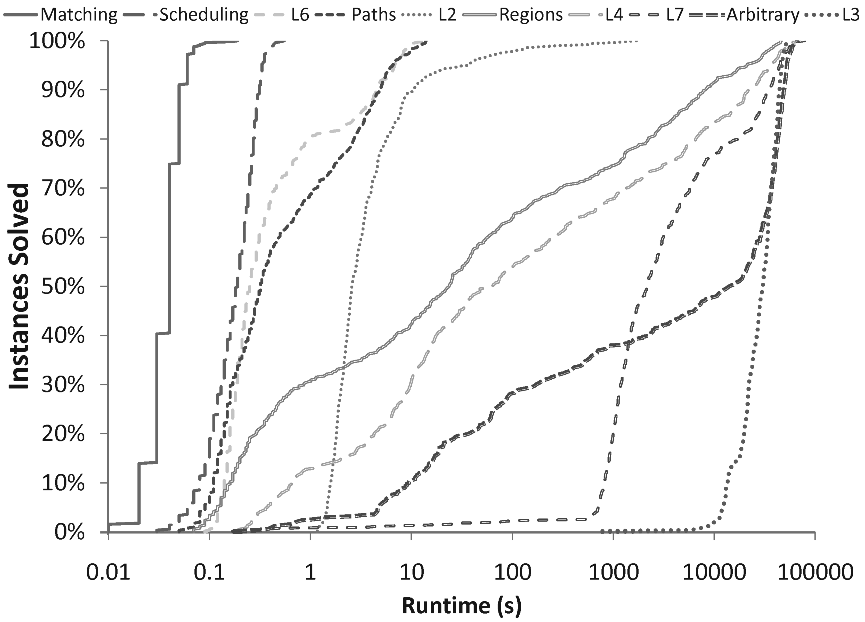

FIG. 4. Gross hardness, 1000 bids/256 goods.

for building models, the validation set for model optimization (e.g., setting the ridge parameter, determining which subset to use when performing subset selection), and the test set for evaluating the quality of our models.

Figure 4 shows the results of our CPLEX runs for each distribution on problems with 256 goods and 1000 nondominated bids, indicating the percentage of instances from each distribution that were solved within a given amount of time. Each instance of each distribution had different parameters, each of which was sampled from a range of values (see our data online for details). Figure 5 presents the corresponding cumulative density function (CDF) for the variable-size data set. Note the log scale on the x axis. We can see that several of the CATS distributions were quite easy for CPLEX, and that others varied from easy to hard. It is interesting that most distributions had instances that varied in hardness by several orders of magnitude, even when all instances had the same problem size.

3.5. STEP 5: BUILDING MODELS

3.5.1. *Linear Models.* We begin by describing the effectiveness of simple linear regression. Besides serving as a baseline, insights into factors that influence hardness gained from a linear model are useful even if other, more accurate models can also be built. As discussed in Section 2.5.2, we chose to use the (base-10) logarithm of CPLEX running time as our response variable—that is, to use an exponential model—rather than using absolute running time. Note that if we had tried to predict absolute running times then the model would have been penalized very little for dramatically mispredicting the running time of very easy instances, and would have been penalized heavily for slightly mispredicting the running time of the hardest instances. Because of this motivation, we do not apply the inverse transformation $h(y)$ when reporting error metrics.

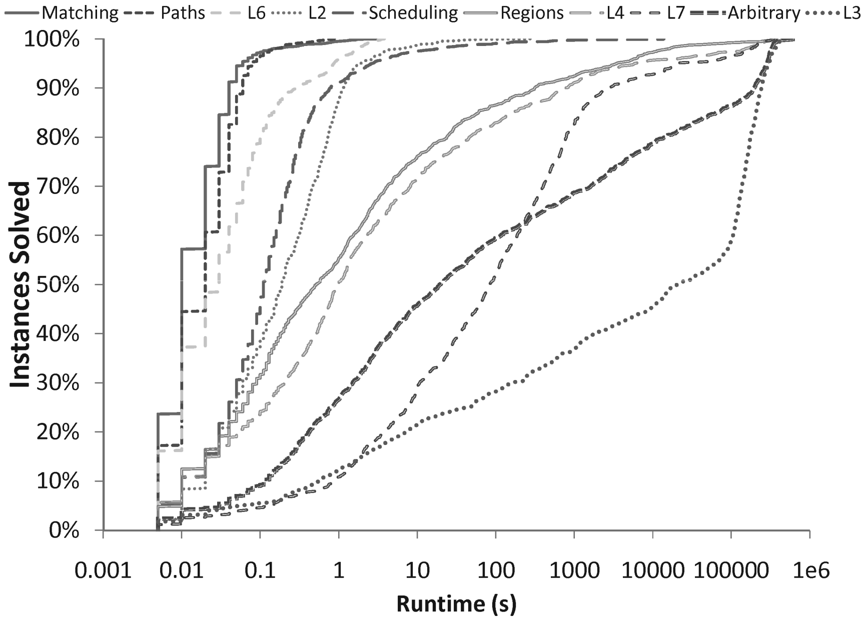

FIG. 5. Gross hardness, variable size.

TABLE I. LINEAR REGRESSION: ERRORS AND ADJUSTED R^2 (TEST DATA)

Data point	Mean Abs. Err.	RMSE	Adj-R^2
1000 bids/256 goods	0.3349	0.5102	0.9114
1000 bids/144 goods	0.3145	0.4617	0.8891
2000 bids/64 goods	0.2379	0.3670	0.9113
Variable Size	0.8673	1.1373	0.7076

In Table I, we report both RMSE and mean absolute error, since the latter is often more intuitive. A third measure, adjusted R^2, is the fraction of the original variance in the response variable that is explained by the model, with an adjustment to correct for inaccuracy arising in the case where the number of training examples is comparable to the number of free parameters in the model. Adjusted R^2 is a measure of fit to the training set and cannot detect overfitting; nevertheless, it can be an informative measure when presented along with test set error. Figure 6 shows the cumulative distribution of squared errors on the test set for the 1000 bids, 256 goods data set. The horizontal axis represents the squared error, and the vertical axis corresponds to the fraction of instances that were predicted with error not exceeding the x-value. Figure 7 shows a scatterplot of predicted $\log_{10}$ runtime vs. actual $\log_{10}$ runtime. Figures 8 and 9 show the same information for the variable-size data. We can see from these figures that most instances are predicted very accurately, and few instances are dramatically mispredicted. Overall, runtimes for 96% of the data instances in our fixed-size test set were predicted to the correct order of magnitude (i.e., with an absolute error of less than 1.0), even without knowing the distribution from which each instance was drawn. On the variable-size data, 64% of instances were predicted within the correct order of magnitude. Finally, observe the vertical banding in the left sides of the scatter plots in Figures 7 and 9, and likewise the

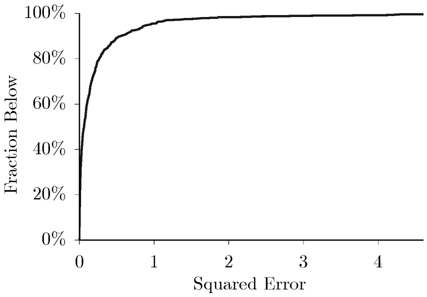

FIG. 6. Linear regression: Squared error (test data, 1000 bids/256 goods).

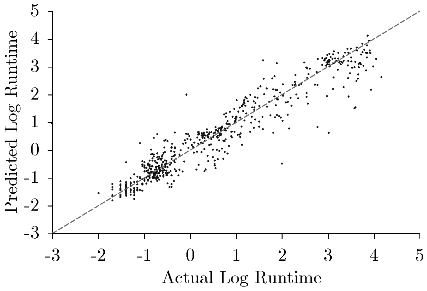

FIG. 7. Linear regression: Prediction scatterplot (test data, 1000 bids/256 goods).

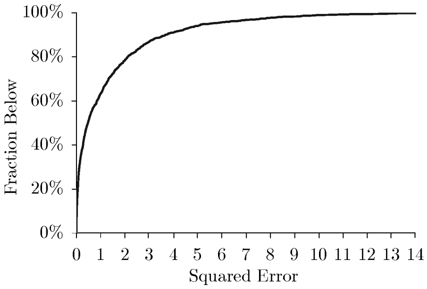

FIG. 8. Linear regression: Squared error (test data, variable size.)

discrete steps in the leftmost CDFs in Figure 4 and 5. This effect is due to the limited precision of the CPU-time process timer used to measure runtimes.

In addition to the root-mean-squared-error-minimizing linear models just discussed, we also investigated the use of other error metrics. In particular, we have considered linear Support Vector Machine (SVM) regression (which minimizes ℓ_1 error), and lasso and ridge regression (shrinkage techniques that express a preference for models with smaller coefficients). None of these approaches achieved significantly better performance than RMSE-minimizing linear models.

3.5.2. *Nonlinear Models.* Of course, there is no reason to presume that the relationship between running time and our features should be linear. We thus investigated various approaches for nonlinear regression. Our overall conclusion was that quadratic regression achieved good performance while requiring relatively little computational effort and while yielding models that were relatively amenable to

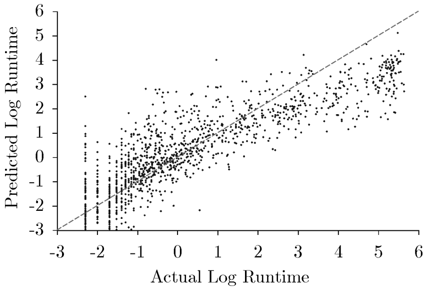

FIG. 9. Linear regression: Prediction scatterplot (test data, variable size).

analysis. Before discussing quadratic regression further, we will begin by describing the other approaches that we considered.

First, we again tried ridge and lasso regression, and again found that at their optimal parameter settings (determined using our validation set) they offered no improvement. However, as mentioned above we did use a very small ridge parameter (10^{-6}) in our quadratic regression, in order to improve numerical stability. Second, we tried Multivariate Adaptive Regression Splines (MARS) [Friedman 1991]. MARS models are linear combinations of the products of one or more basis functions, where basis functions are the positive parts of linear functions of single features. Unfortunately, MARS models can be unstable and difficult to interpret. For us, they also offered no benefit over quadratic models: the RMSE on our MARS models differed from the RMSE on our second-order model only in the second decimal place. Finally, we tried SVM regression with a second-order polynomial kernel. Even after optimizing the SVM parameters through experiments on our validation set, we again achieved no significant difference in performance as compared to quadratic models. SVM regression has several disadvantages when compared to linear regression: models are more computationally expensive to train, the method's behavior is relatively sensitive to parameters, and our subset selection techniques (see Section 4) are complicated by the use of kernels. Finally, we briefly experimented with SVM regression using higher-order polynomial kernels, but did not find them to be useful.[15]

In order to construct quadratic models, we first computed all pairwise products of features, including squares, and also retained the original features, for a total of 495 features in all data sets. However, this introduced a lot of redundant features, as even most of the original (linear) features were highly correlated. We thus used the preprocessing step discussed in Section 2.5.2, with an adjusted R^2 threshold of $\tau = 0.99999$. Even with such a conservative threshold, this process eliminated a lot of features. On the 1000 bids, 256 goods data set we were left with 353 features out of 495, and on the variable-size data set with 423 out of 495. Overall, this preprocessing step yielded more accurate and numerically stabler models.

[15] As mentioned earlier, more recent work on SAT has further experimented with nonlinear, nonparametric methods such as Gaussian processes [Hutter et al. 2006] and random forests of regression trees. Again, while these methods have various advantages and disadvantages, so far we have not observed any of them to yield significant improvements in prediction accuracy when the training data set is large.

TABLE II. QUADRATIC REGRESSION: ERRORS AND ADJUSTED R^2 (TEST DATA)

Data point	Mean Abs. Err.	RMSE	Adj-R^2
1000 bids/256 goods	0.2099	0.3122	0.9653
1000 bids/144 goods	0.2270	0.3548	0.9482
2000 bids/64 goods	0.1693	0.3032	0.9563
Variable Size	0.3753	0.5354	0.9473

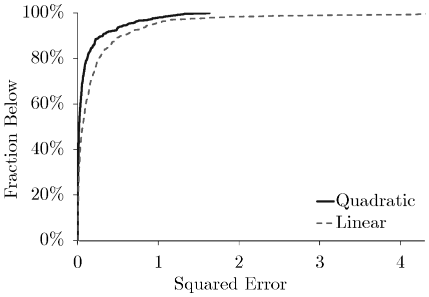

FIG. 10. Quadratic regression: Squared error (test data, 1000 bids/256 goods).

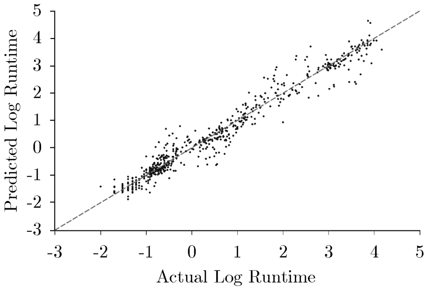

FIG. 11. Quadratic regression: Prediction scatterplot (test data, 1000 bids/256 goods).

For all of our data sets quadratic models yielded considerably better error measurements on the test set and also explained nearly all the variance in the training set, as shown in Table II. As above, Figures 10, and 12 show cumulative distributions of the squared error, and Figures 11 and 13 show scatterplots of predicted $\log_{10}$ runtime vs. actual $\log_{10}$ runtime for fixed-size and variable-size data. Comparing these figures to Figures 6, 7, 8, and 9 confirms our judgment that quadratic models are substantially better overall. The cumulative distribution curves lie well above the corresponding curves for linear models. For example, quadratic models would classify 98% of test instances in the fixed-size data set and 93% of instances in the variable-sized data set to within the correct order of magnitude—a dramatic improvement over linear models.

4. *Analyzing Hardness Models*

The results summarized above demonstrate that it is possible to learn a model of our features that accurately predicts CPLEX's running time on novel instances. For some applications (e.g., predicting the time it will take for an auction to clear; building an algorithm portfolio) accurate prediction is all that is required. For other applications it is necessary to *understand* what makes an instance empirically hard.

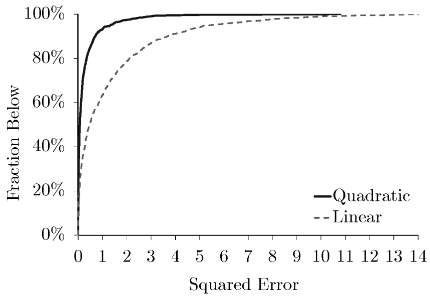

FIG. 12. Quadratic regression: Squared error (test data, variable size).

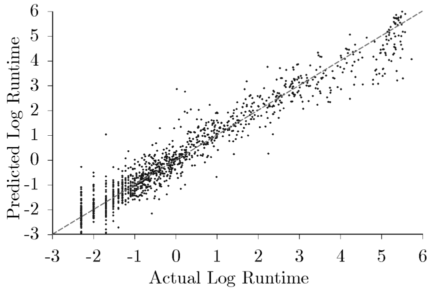

FIG. 13. Quadratic regression: Prediction scatterplot (test data, variable size).

In this section, we set out to describe how our models may be interpreted; the following section applies these ideas to our WDP case study.

A key question in explaining what makes a hardness model work 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 if $|w_i| \gg |w_j|$ then f_i must be more important than f_j. This can be misleading for two reasons. First, features may have different ranges, though this problem can be mitigated by normalization. More fundamentally, when two or more features are highly correlated, models can include larger-than-necessary coefficients with different signs. For example, suppose that we have two identical and completely unimportant features $f_i \equiv f_j$. Then all models with $w_i = -w_j$ are equally good, even though in some of them $w_i = 0$, while in others $|w_i|$ is arbitrarily large. This problem can be mediated through the use of regularization techniques; however, it cannot be entirely eliminated, especially when many features are strongly but imperfectly correlated.

A better approach is to force models to contain fewer variables. The idea here is that the best low-dimensional model will involve only relatively uncorrelated features, because adding a feature that is very correlated with one that is already present will yield a smaller marginal decrease in the error metric. 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. We considered three heuristic methods: *forward* *selection*, *backward elimination*, and *sequential replacements*. More sophisticated heuristic search techniques could also be used to improve on these relatively simple algorithms.[16] Since none of these techniques is guaranteed to find the optimal subset, they can be combined together by running all and keeping the model with the smallest validation-set error.

Forward selection starts with an empty set, and greedily adds the feature that, combined with the current model, makes the largest reduction to validation-set error.[17] Backward elimination starts with a large model and greedily removes the features that yields the smallest increase in validation set error. Sequential replacement is like forward selection, but also has the option to replace a feature in the current model with an unused feature.[18]

Each *step* of each subset selection technique must build a number of models linear (forward, backward) or quadratic (sequential) in the total number of features. Thus, it is worthwhile to explore faster ways of building these models. The Shermann-Morrison formula, also known as the matrix inversion lemma, states that

$$(A + uv^T)^{-1} = A^{-1} - \frac{1}{1 - v^T A^{-1} u}(A^{-1}u)(v^T A^{-1})^T.$$

This means that if we can express the new matrix to be inverted in terms of one or more rank-one updates of a matrix whose inverse we already know, the new inverse can be computed in quadratic rather than cubic time. In fact, we can do so for all of our subset selection approaches: they involve repeatedly inverting symmetric matrices that differ in exactly one row and column because of the insertion and/or removal of one feature. The new matrix can be constructed from the original in two (forward, backward) or four (sequential) rank-one updates.

Once a model with a small number of variables has been obtained, we can evaluate the importance of each feature to that model by looking at each feature's *cost of omission*, following Friedman [1991]. That is, to evaluate *score*(f_i) we can train a model that omits f_i, and report the resulting increase in (validation set) prediction error compared to the model that includes f_i. Notice that this would not work in the presence of highly correlated features: if $f_i \equiv f_j$ and is very useful, then dropping either one will not result in any increase in the error metric, leading us to believe that both features are useless.

It is very important to recognize what it means for our techniques to identify a variable as "important." If a set of variables X is identified as the best subset of a given size, and this subset has a RMSE that is close to the RMSE of the complete model, this indicates that the variables in X are *sufficient* to approximate the performance of the full model—useful information, since it means that we can explain an algorithm's empirical hardness in terms of this small set of features. It must be stressed, however, that this does not amount to an argument that choosing

[16] When we applied our methodology to SAT [Nudelman et al. 2004b] we also used the LAR algorithm [Efron et al. 2004]. LAR 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. Even more recently, Xu, Hutter, Hoos, and Leyton-Brown [2008] implemented a local-search-based subset selection algorithm.

[17] Of course, everywhere we propose the use of a validation set, cross-validation is also an option.

[18] For a detailed discussion of techniques for selecting relevant feature subsets and for comparisons of different definitions of "relevant," focusing on classification problems, see Kohavi and John [1997].

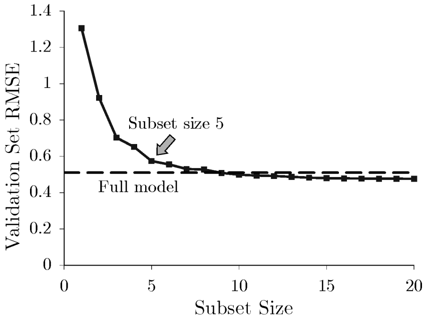

FIG. 14. Linear regression: Subset size vs. RMSE (validation data; 1000 bids/256 goods).

the subset X is *necessary* for good performance in a subset of size k. Indeed, as it is often the case that many variables are correlated in complex ways, there will often be other very different sets of features that would achieve similar performance. Furthermore, since our subset selection techniques are heuristic, we are not even guaranteed that X is the globally best subset of its size. Thus, we can conclude that features are important when they are *present* in small, well-performing subsets, but we must be careful not to conclude that features that are *absent* from small models are unimportant.

5. *Analyzing Hardness Models: WDP Case Study*

In this section, we discuss the application of our methodology for analyzing empirical hardness models to our WDP case study. Despite the fact that we observed that quadratic models strongly outperformed linear models in Section 3.4, we analyze both sets of models here. This is because we can learn different things from the different models. In particular, linear models can sometimes lead to simpler intuitions since they do not depend on products of features.

5.1. LINEAR MODELS. Figure 14 shows the RMSEs of the best subsets containing between 1 and 20 features for linear models on the 1000 bids, 256 goods data set. To find the best subset of size k, we evaluated the validation-set RMSE of the three k-variable models produced by forward selection, backward elimination and sequential replacements, and kept the model with the smallest RMSE. The dashed line represents the test-set RMSE of the full linear model. As discussed in Section 4, our use of heuristic subset selection techniques means that the subsets shown in Figure 14 are likely not the RMSE-minimizing subsets of the given sizes. Nevertheless, we can still conclude that subsets of these sizes are *sufficient* to achieve the accuracies shown here. We chose to examine the model with five features because it was the first for which adding another feature did not cause a large decrease in RMSE. This suggests that the rest of the features were relatively highly correlated with these five, or were relatively unpredictive of running time. Figure 15 shows the features in this model, the signs of the corresponding coefficients in the regression model (showing whether an increase in the feature led to an increase or decrease in the predicted runtime), and their respective costs of omission, expressed in RMSE. Figures 16 and 17 demonstrate the results of subset selection for linear models on our variable-size data set. For this data set, we also picked a five-variable model, to make it easier to compare our findings across data sets.

Feature	±	Cost
BG maximum degree	+	.550
integer slack ℓ_1 norm	+	.339
BGG minimum good degree	+	.118
BGG maximum good degree	−	.091
BGG stdev of bid degree	−	.077

FIG. 15. Linear regression: Feature, sign of feature coefficient, and cost of omission for subset size 5 (validation data; 1000 bids/256 goods).

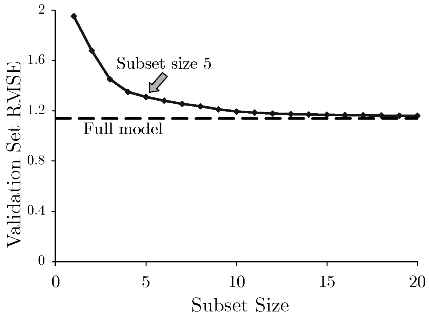

FIG. 16. Linear regression: Subset size vs. RMSE (validation data, variable size).

Feature	±	Cost
BG third quartile of degree	+	.489
BG average clustering	−	.337
number of goods	+	.174
BGG maximum good degree	−	.078
integer slack ℓ_∞ norm	+	.041

FIG. 17. Linear regression: Feature, sign of feature coefficient, and cost of omission for subset size 5 (test data, variable size).

The most overarching conclusion we can draw is that features based on node degrees and on the LP relaxation are important. Node degree features describe the constrainedness of the problem. Generally, one might expect that very highly constrained problems would be easy, since more constraints imply a smaller search space. However, our experimental results show that CPLEX takes a long time on such problems. We conjecture that either CPLEX's calculation of the LP bound at each node becomes much more expensive when the number of constraints in the LP increases substantially, or that the accuracy of the LP relaxation decreases (along with the number of nodes that can be pruned). In either case, this cost overwhelms the savings that come from searching in a smaller space. The node degree statistics describe the number of constraints in which each variable is involved; they indicate how quickly the search space can be expected to narrow as bids are assigned to the allocation. The most important features in both models relate to the degree of the Bid Graph. These features each describe the numbers of other bids that will be ruled out every time a bid is added to an allocation. Features describing the good degree of the Bid-Good Graph also appear in both models. These features are similar to the Bid Graph features; they describe the number of bids that will be ruled out when a single *good* (rather than bid) is allocated.

Norms of the linear programming slack vector also appear in both models. This is very intuitive; CPLEX employs an LP relaxation as its guiding heuristic for the branch-and-bound search. First, the easiest problems can be completely solved by LP, yielding a norm of 0; the norms are close to 0 for problems that are almost completely solved by LP (and hence usually do not require much search to resolve), and larger for more difficult problems. Second, the integrality gap represents the quality of the heuristic and, consequently, the extent to which the branch-and-bound search space will be pruned.

A few other features were also selected. The clustering coefficient features measure the extent to which variables that conflict with a given variable also conflict with each other, another indication of the speed with which the search space will narrow as variables are assigned.[19] This feature gives an indication of how local the problem's constraints are. The final remaining feature in our fixed-size model measures variation in BGG bid node degrees. Observe in the same model that the presence of features measuring maximum and minimum BGG node degrees, with opposite signs in the features' coefficients, amounts to another measure of variation, now of the degrees of *good* nodes.

The final feature in our variable-size model is the number of goods. This is not surprising, since the worst-case complexity of the WDP scales exponentially in the number of goods but only polynomially in the number of bids [Sandholm 2002]. Experimental evidence confirms that the number of goods is often a more important parameter for empirical hardness, though it shows that the number of bids is also important in practice.

We take this opportunity to offer a comment about incorporating problem size features into empirical hardness models. Though problem size parameters are important, we have found overall that they are less strongly correlated with empirical hardness than one might suppose. When we began studying the effect of input size on runtime, we conducted scaling experiments, in which we observed that for all input sizes that we could reasonably attempt to solve, runtimes varied for many orders of magnitude—from less than a second to days or even weeks. It thus appears that for modern WDP algorithms, problem structure is paramount and asymptotic complexity does not strongly manifest itself until size is increased well beyond what is feasible. In particular, this led us to reject an earlier approach in which we considered the use of alternative hypothesis spaces that give special treatment to input size variables. Instead, we now treat these features as being fundamentally the same as structural features. We do note that the same pattern may not hold in all problem domains.

5.2. NONLINEAR MODELS. We now consider second-order models. We ran forward selection and sequential replacement up to models with 40 features, and ran backward elimination starting with the 40-feature model identified by forward selection. Figure 18 describes the best subsets containing between 1 and 40 features for second-order models on the fixed-size data set, with the dashed line indicating the test-set RMSE of the full quadratic model. We observe that allowing interactions between features dramatically improved the accuracy of our very small subset models; indeed, our six-feature quadratic models outperformed the full linear

[19] We also observed that this feature was important when building empirical hardness models for SAT [Nudelman et al. 2004b]. We thank Rámon Béjar for providing code for calculating the clustering coefficient.

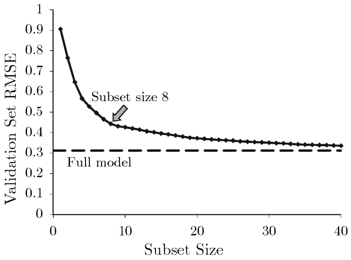

FIG. 18. Quadratic regression: Subset size vs. RMSE (validation data, 1000 bids/256 goods).

Feature	±	Cost
BGG avg bid deg * int slack ℓ_1 norm	+	.258
BG max deg * num goods	+	.148
BGG max good deg * num goods	−	.085
BGG max bid deg * int slack ℓ_2 norm	−	.063
BG avg clustering * stdev($\frac{\text{price}}{\sqrt{\text{bid size}}}$)	−	.041
(BGG max good deg)^2	+	.015
BGG min good deg * stdev($\frac{\text{price}}{\sqrt{\text{bid size}}}$)	+	.012
BGG min bid deg * $\frac{\text{BG avg clustering}}{\text{BG avg min path length}}$	−	.003

FIG. 19. Quadratic regression: Feature, sign of feature coefficient, and cost of omission for subset size 8 (validation data, 1000 bids/256 goods).

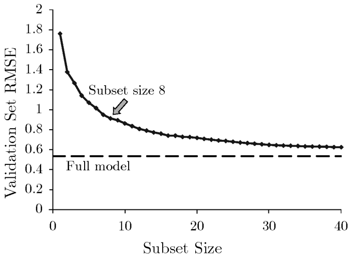

FIG. 20. Quadratic regression: Subset size vs. RMSE (validation data, variable size).

models (see error measurements for the linear models in Table I). Figures 20 and 21 show a similar picture for our variable-size data set. Note that on this data set a five-feature quadratic model outperformed the full linear model (again, see Table I). As above, we selected the smallest subset size at which the marginal benefit of adding another feature was small for both models; in this case, we elected to examine eight-variable models. Figures 19 and Figure 21 show the costs of omission for these models, considering the fixed-size and variable-size data sets respectively.

We note that nearly all the selected features were second-order: only one of the sixteen was not. (Recall that all first-order features remained available for the models to select.) However, we observed many of the same trends as in our linear models. In particular, the features that were important to the linear models tended to be chosen again in our quadratic models: in both cases, four of the five features were chosen, and the features that were not were the least important and second-least important features in the linear models. More broadly, node degree statistics and linear programming norms remained very important. Indeed, the two models had only one and two features respectively that *did not* involve either of these

Feature	±	Cost
num bids * int slack ℓ_1 norm	+	.393
BG max degree * int slack ℓ_∞ norm	+	.196
BG avg clustering	−	.168
BG avg eccentricity * BG stdev of clustering	−	.135
BG min degree * num bids	+	.085
BG degree 3rd quartile * num goods	+	.071
BG avg clustering * int slack ℓ_2 norm	−	.052
BGG stdev of good deg * int slack ℓ_∞ norm	−	.036

FIG. 21. Quadratic regression: Feature, sign of feature coefficient, and cost of omission for subset size 8 (validation data, variable size).

elements. All three of these features involved the clustering coefficient, which we also observed in our linear models.

We also observed some new features that did not appear in our linear models. First, problem size features played a different role. Most strikingly, the number of bids, once allowed to interact with other features, became much more prominent in our variable-size model. Indeed, it became part of the most important feature. It appears that the number of bids was useful as a scaling factor for other features that had previously been important. The number of goods also appeared in the variable-size model, but was less prominent than in the linear case. Interestingly, though, the number of goods was also selected in the *fixed-size* model. Note that multiplying a feature by a constant has no effect on the predictive power of a linear regression model; the feature's coefficient can simply be adjusted. However, recall that the number of goods varied slightly in the CATS distributions. We conjecture that with an appropriate coefficient the (e.g.) `BG max deg * num goods` feature was just as useful as `BG max deg` for instances from distributions under which the number of goods did not vary, but captured some additional useful information about problem size on CATS instances.

Second, our price-based features appeared in the fixed-size model, but not in the variable-size model. Overall, we were surprised that these features were not more important, given that they were the only ones that explicitly considered the optimization problem's objective function. In the end, it appears that combinatorial structure has a stronger effect on empirical hardness for WDP.

Finally, we saw features that considered global measures of constrainedness. In particular, both our path length and eccentricity features appeared, though neither played a prominent role.

Based on the discussion so far, we can now understand the two most important features in each of our eight-feature quadratic models. Each of these features is the product of two of the following: a node degree feature, a problem size feature, and an LP integrality slack norm feature. Furthermore, each of these features has a positive coefficient. We have previously explained why more highly-constrained problems, larger problem sizes, and larger LP integrality slack norms each lead to problems that are harder for CPLEX to solve. A feature that is the product of two of these terms takes a large value only when both components are large. This product can therefore yield a powerful prediction of an instance's hardness.

6. *Boosting as a Metaphor for Algorithm Design*

We now turn to the consideration of practical uses to which we can put empirical hardness models. The key phenomenon we will consider here is that, although some algorithms are better than others on average, there is rarely a best algorithm for a given problem. Instead, it is often the case that different algorithms perform well on different problem instances. Not surprisingly, this phenomenon is most pronounced among algorithms for solving NP-complete problems, because runtimes for these algorithms often vary substantially from instance to instance. When algorithms exhibit high runtime variance, it can be difficult to decide which algorithm to use. Rice [1976] dubbed this the "algorithm selection problem." In the nearly three decades that have followed, the issue of algorithm selection has failed to receive widespread study, though of course some excellent work does exist. (We discuss much of this work in Section 6.2.1.) The most common approach to algorithm selection is measuring different algorithms' performance on a given problem distribution and then using only the algorithm that had the lowest average runtime. This approach, which we dub "winner-take-all," has driven recent advances in algorithm design and refinement. However, it has resulted in the neglect of many algorithmic ideas that, while uncompetitive on average, offer excellent performance on particular classes of problem instances. Our consideration of the algorithm selection literature, and our dissatisfaction with the winner-take-all approach, has led us to ask the following two questions. First, what general techniques can we use to perform per-instance (rather than per-distribution) algorithm selection? Second, if we reject the notion of winner-take-all algorithm evaluation, how ought novel algorithms to be evaluated? Taking the idea of boosting from machine learning as our guiding metaphor, we strive to answer both questions.

6.1. The Boosting Metaphor.

Boosting is a machine learning paradigm due to Schapire [1990] and widely studied since. Although our article does not make use of any technical results from the boosting literature, we take our inspiration from the boosting philosophy. Stated simply, boosting is based on two insights:

(1) Poor classifiers can be combined to form an accurate ensemble when the classifiers' areas of effectiveness are sufficiently uncorrelated.
(2) New classifiers should be trained on problems on which the current aggregate classifier performs poorly.

We argue that algorithm design should be informed by two analogous ideas:

(1) Algorithms with high average running times can be combined to form an algorithm portfolio with low average running time when the algorithms' easy inputs are sufficiently uncorrelated.
(2) New algorithm design should focus on problems on which the current algorithm portfolio performs poorly.

Although it is helpful, our analogy to boosting is clearly imperfect. One key difference lies in the way components are aggregated in boosting: classifiers can be combined through majority voting, whereas the whole point of algorithm selection is to run only a single algorithm. We instead advocate the use of learned models of runtime as the basis for algorithm selection, which leads to another important difference. It is not enough for the easy problems of multiple algorithms to be uncorrelated; models must also be accurate enough to reliably recommend against the slower algorithms on these uncorrelated instances. Finally, while it is impossible to improve on correctly classifying an instance, it is almost always possible to solve a problem instance more quickly. Thus improvement is possible on easy instances as well as on hard instances. The analogy to boosting holds in the sense that focusing on hard regions of the problem space increases the potential gain in terms of reduced average portfolio runtimes.

6.2. Boosting Step 1: Building Algorithm Portfolios. The first part of this article was devoted to arguing that it is possible to build accurate algorithm-specific models of the empirical hardness of given distributions of problem instances. Given these techniques, we advocate the following straightforward method for building portfolios of multiple algorithms.

(1) Train an empirical hardness model for each algorithm, following the methodology given in Section 2.
(2) Given an instance:
    (a) Compute feature values.
    (b) Predict each algorithm's running time using the empirical hardness models learned in Step 1.
    (c) Run the algorithm predicted to be fastest.

We should clarify our use of the term *portfolio*. This term was first used by Huberman et al. [1997] and Gomes and Selman [2001] to describe a strategy for running a set of algorithms in parallel.[20] The term has since also been used to describe any strategy that combines multiple, independent algorithms to solve a single problem instance. The argument behind this broadening of the term—which we favor—is that all such methods exploit lack of correlation in the performance of several algorithms to obtain improved overall performance.

To facilitate precise characterization of algorithm portfolios in this broader sense, we offer some terminology, following Xu, Hutter, Hoos, and Leyton-Brown [2008, pp. 567–568]. A (a, b)-of-n portfolio is a set of n algorithms and a procedure for selecting among them with the property that if no algorithm terminates early, at least a and no more than b algorithms will be executed. The term a-of-n portfolio is shorthand for an (a, a)-of-n portfolio, and the term n-portfolio is shorthand for an n-of-n portfolio. Portfolios also differ in how solvers are run after being selected. When all algorithms are run concurrently, a portfolio is said to be *parallel*. When one algorithm's execution always begins after another's ends, the portfolio is *sequential*. Otherwise, the portfolio is said to be *partly sequential*. Thus, the algorithm portfolios of Huberman et al. [1997] and Gomes and Selman [2001] can be described as parallel n-portfolios. In contrast, the empirical-hardness-model-based portfolios discussed above are 1-of-n portfolios. (Since only one algorithm is selected, the sequential/parallel distinction is moot in this case.) Richer combinations are possible and indeed practical; for example, Xu et al. [2008] extended the techniques introduced here to build sequential 3-of-n portfolios that won in several categories of the 2007 SAT competition.

Overall, while we will show experimentally that our portfolios can dramatically outperform the algorithms of which they are composed, the simplicity of our techniques is somewhat deceptive. We discuss other approaches from the literature and compare them with our own in Section 6.2.1. Following this discussion, we consider

[20] Indeed, this terminology is consistent with the original notion of a portfolio from finance, where risk is managed through the purchase of multiple different securities.

ways of enhancing our techniques for building empirical hardness models for use in selecting among algorithms in an algorithm portfolio.

6.2.1. *Alternative Approaches to Algorithm Selection.* Much past work has observed that since algorithm performance can vary substantially across different classes of problems, performance gains can be achieved by selecting among these algorithms on a per-instance basis. As mentioned above, Rice [1976] was the first to formalize algorithm selection as a computational problem, framing it in terms of function approximation. Broadly, he identified the goal of selecting a mapping $S(x)$ from the space of instances to the space of algorithms, to maximize some performance measure $\mathrm{perf}(S(x), x)$. Rice offered few concrete techniques, but all subsequent work on algorithm selection can be seen as falling into his framework.

In what follows, we explain our choice of methodology by relating it to other approaches for algorithm selection that have been proposed in the literature.

6.2.1.1. *Regression Approaches.* Brewer [1994, 1995] proposed an approach very similar to our own in the context of parallel computing. Specifically, he used linear regression models to predict the runtime of different implementations of portable, high-level libraries for multiprocessors, with the goal of automatically selecting the implementation with the best predicted runtime on a novel architecture. Key similarities are the use of linear regression models to predict runtime based on empirical measurements of black-box algorithms, and algorithm selection by choosing the algorithm with the best predicted performance. The key difference is that he focused on sub-quadratic-time rather than NP-complete problems. (Specifically, he studied sorting and stencil computations, with respective asymptotic complexities of $O(n \log n)$ and $O(n^{1.5})$.) These problems do not exhibit the extreme runtime variability we have observed for WDP. Probably for this reason, Brewer was able to perform very accurate algorithm selection by relying only on a handful of features describing problem size and machine architecture, and making use of a simpler and less flexible regression framework than we found necessary.

Lobjois and Lemaître [1998] selected among several simple branch-and-bound algorithms based on a method for estimating search tree size due to Knuth [1975]. Although they use a sampling-based rather than a model-based approach, we can broadly interpret their work as performing regression-based algorithm selection. This work is similar in spirit to our own; however, their prediction is based on a single feature and works only on a particular class of branch-and-bound algorithms.

6.2.1.2. *Parallel Execution.* A number of authors have proposed the use of parallel portfolios. We have already mentioned Huberman et al. and Gomes and Selman; more recent examples include Gagliolo and Schmidhuber [2006] and Streeter et al. [2007]. There are good reasons to pursue parallel portfolios: in particular, they can be simpler to build, and it may be difficult or impossible to build models that describe the effect of variables such as random seed on an algorithm's performance. Indeed, particularly because of this latter argument, we discuss the extension of our own portfolio approach to the construction of parallel portfolios in Section 6.2.3.

However, parallel portfolios are often outperformed by our 1-of-*n* portfolios. We focus here on the simple case of parallel portfolios that assign the same CPU share to each algorithm. Even in this case, to make sense of our claim, we must first decide how to compare running times between these two approaches. While it is often true that additional processors are readily available, it is also often the case that these processors can be put to uses besides running different algorithms in parallel, such as parallelizing a single search algorithm or solving multiple problem instances at the same time. We therefore believe that meaningful comparisons of running time between parallel and non-parallel portfolios require that computational resources be fixed, with parallel execution modeled as ideal (no-overhead) task swapping on a single processor. Let $t^*(x)$ be the time it takes to run the algorithm that is fastest on instance x, and let n be the number of algorithms. A parallel n-portfolio that assigns an equal share to every algorithm for each instance x (a parallel n-portfolio) will always take time $nt^*(x)$. The expected value of $nt^*(x)$ can be much smaller than the expected run-time of the algorithm that would be chosen under winner-take-all algorithm selection. Nevertheless, we will see in Section 7.1 that our portfolio approach can do even better than this. (For example, we will show on our fixed-size WDP data that such parallel execution has roughly the same average runtime as winner-take-all algorithm selection—we have three algorithms and CPLEX is three times slower than the optimal portfolio—while our techniques achieve running times of roughly $1.09t^*(x)$.)

6.2.1.3. *Classification.* Since algorithm selection is fundamentally discriminative—it entails choosing among algorithms to find one that will exhibit minimal runtime—classification is an obvious approach to consider. Indeed, our approach of fitting regression models and then choosing the algorithm predicted to be fastest can be understood as a classification algorithm. However, more standard classification algorithms (e.g., decision trees) can also be used to learn which algorithm to choose given features of the instance and labelled training examples. For example, such an approach was taken by Gebruers et al. [2005], who investigated the use of case-based reasoning and decision trees to select among a discrete set of constraint programming strategies in a scheduling domain; some of the same authors also did earlier work in a similar vein [Gebruers et al. 2004; Gebruers and Guerri 2004]. Guo and Hsu [2007] considered the use of decision trees, naive Bayes models and Bayes net models (all classification methods) for the construction of algorithm portfolios consisting of both complete and incomplete algorithms for the MPE problem in Bayesian networks.

It is perfectly reasonable in principle to explicitly use classification algorithms to perform algorithm selection. However, a problem often does arise in practice. Most off-the-shelf implementations of classification algorithms use the wrong error metric for our domain: they penalize misclassifications equally regardless of their cost. We want to minimize a portfolio's average runtime, not its accuracy in choosing the optimal algorithm. Thus, we should penalize misclassifications more when the difference between the run-times of the chosen and fastest algorithms is large than when it is small. This is just what our approach does.

A second classification approach entails dividing running times into two or more bins, predicting the bin that contains the algorithm's runtime, and then choosing the best algorithm. For example, Horvitz et al. [2001] and Ruan et al. [2002] used classification to predict the run-time of CSP and SAT solvers with inherently high runtime variance (heavy tails), albeit during runs. Despite its similarity to our portfolio methodology, this approach suffers from the same problem as described above. First, the learning algorithm described in this work did not use an error function that penalized large misclassifications (off by more than one bin) more heavily than small misclassifications (off by one bin). Second, this approach is unable to discriminate between algorithms when multiple predictions fall into the same bin. Finally, since run-time is a continuous variable, class boundaries are artificial. Instances with runtimes lying very close to a boundary are likely to be misclassified even by a very accurate model, making accurate models harder to learn.

Finally, we note that while our approach may be understood as performing classification, it has an advantage over other classification algorithms even if they do use a cost-sensitive error metric. Our approach makes a separate prediction for each algorithm and then chooses the best one. This means that once an empirical hardness model has been built and validated for a given algorithm, that model never needs to be changed. If new algorithms are to be added, it is not necessary to learn an entirely new classifier: all that is required is to learn a good empirical hardness model for the new algorithm.

6.2.1.4. *Markov Decision Processes.* Lagoudakis and Littman [2000, 2001] worked within the MDP framework and concentrated on recursive algorithms (e.g., sorting, SAT), sequentially solving the algorithm selection problem on each subproblem. This work demonstrated encouraging results; however, its generality was limited by several factors. First, the use of algorithm selection at each stage of a recursive algorithm can require extensive recoding, and may simply be impossible with the "black-box" commercial or proprietary algorithms that are often among the most competitive. Second, solving the algorithm selection problem recursively requires that value functions be very inexpensive to compute; for the WDP we found that more computationally expensive features were required for accurate predictions of run-time. Finally, these techniques can be undermined by non-Markovian algorithms, such as those using clause learning, taboo lists, caching or other forms of dynamic programming. Of course, our approach could also be characterized as an MDP; we do not do so as the framework is redundant in the absence of a sequential decision-making problem.

Recent work by Carchrae and Beck [2005] can also be seen as falling into the MDP framework, in the sense that it involves the use of machine learning to make sequential decisions in order to maximize an objective function. This work considers both complete and incomplete algorithms for a scheduling problem, and attempts to learn a classifier that predicts the solution quality that each algorithm will be able to achieve by a given cutoff time, based on initial probing runs of each algorithm. This work considers both the offline algorithm selection problem and the online problem in which fixed resources must be dynamically allocated between the algorithms based on their performance. The authors emphasize that their techniques depend only on "low-knowledge" features of the input instance.

6.2.1.5. *Experts Algorithms.* Another area of research that is somewhat related to algorithm selection is that of "experts algorithms" (see, e.g., de Farias and Megiddo [2004]). The setting studied in this line of work is the following. An agent must repeatedly act in a certain environment. It has access to a number of "experts" that can provide advice about the best action in the current step based on past histories and some knowledge. Thus, in some sense, at each step the agent must solve the algorithm selection problem. This is generally done by estimating past experts' performances, and then choosing the best expert based on these estimates. This estimation step is in principle similar to our use of regression models to predict performance. The main difference between our work and the area of experts algorithms is that in the experts algorithms domain learning and selection are done online. Estimation often takes form of historical averages, and research has concentrated on issues such as the exploration-exploitation tradeoff.

Indeed, the experts approach has been explicitly applied to the problem of learning which heuristics to use in a constraint programming context. For example, Epstein et al. [2002] built an "adaptive constraint engine" in which a set of advisors recommend the use of one heuristic or another, and in which the system learns how to weight these different experts' advice.

6.2.2. *Capping Runs.* The methodology of Section 6.2 requires gathering runtime data for every algorithm on every problem instance in the training set. While the time cost of this step is fundamentally unavoidable under our approach, gathering perfect data for every instance can take an unreasonably long time. We already mentioned (in Section 2.4) that the process of gathering data can be made much easier by capping the runtime of each algorithm at some maximum and recording these runs as having terminated at the captime. When models are intended for use in building an algorithm portfolio, more aggressive capping is possible. For example, if algorithm a_1 is usually much slower than a_2 but in some cases dramatically outperforms a_2, a perfect model of a_1's runtime on hard instances may not be needed to discriminate between the two algorithms. This approach is safe if the captime is chosen so that it is (almost) always significantly greater than the minimum of the algorithms' runtimes. Even if this condition is not quite satisfied, it can still be preferable to sacrifice some predictive accuracy for dramatically reduced model-building time. Note that if any algorithm is capped, it can be dangerous (particularly without the log transformation that occurs in exponential and logistic models) to gather data for any other algorithm without capping at the same time, because the portfolio could inappropriately select the algorithm with the smaller captime. Of course, when different captimes are used for different algorithms, the algorithm(s) that were allowed to run for longer can be "artificially capped" (i.e., have their runtimes rounded down to the smaller captime) after data has been collected.[21]

6.2.3. *Parallel Execution.* As discussed above, when runtime depends heavily on variables such as random seed, it can be difficult to predict. In such cases, parallel execution is likely to outperform a portfolio that chooses a single algorithm. Nevertheless, in such cases it is possible to extend our methodology to incorporate parallel

[21] Work on methods for working with capped runs continued after our submission of this paper for review. Specifically, Xu, Hutter, Hoos & Leyton-Brown [2007b] investigated three methods for building models in the presence of so-called "censored data" (i.e., when the response variable's value is not always observed perfectly, as occurs with capped runs). That paper tested three methods: (i) discarding all capped runs; (ii) treating capped runs as having terminated at the captime, and (iii) building models using an iterative method due to Schmee and Hahn [1979], which arose in the study of survival analysis. Method (iii) produced the best models; thus, we recommend it above method (ii) which is described in Section 6.2.2.

execution, essentially treating different parallel portfolios as different algorithms and modeling their performance.

(1) Train an empirical hardness model for each algorithm, following the methodology from Section 2.
(2) Additionally, train empirical hardness models for one or more new "algorithms", where algorithm i stands as a placeholder for the parallel execution of k_i of the original algorithms. Consider the runtime of algorithm i on a given instance to be k_i times the minimum runtime of i's constituent algorithms.
(3) Given an instance:
    (a) Compute feature values.
    (b) Predict each algorithm's running time using the empirical hardness models learned in Steps 1 and 2.
    (c) Run the algorithm predicted to be fastest.

This approach allows portfolios to choose to task-swap sets of algorithms in parts of the feature space where the minima of individual algorithms' runtimes are much smaller than their means, but to choose single algorithms in other parts of the feature space.

6.2.4. *Smart Feature Computation.* Feature values must be computed before the portfolio can choose an algorithm to run. We expect that portfolios will be most useful when they combine several (worst-case) exponential-time algorithms that have highly uncorrelated runtimes, and that fast polynomial-time features should be sufficient for most models. Nevertheless, on some instances the computation of individual features may take substantially longer than one or even all algorithms would take to run. In such cases it would be desirable to perform algorithm selection without spending as much time computing features, even at the expense of some accuracy in choosing the fastest algorithm.

We begin by partitioning the features into sets ordered by time complexity, $S_1, \ldots, S_l$, with $i > j$ implying that each feature in S_i takes significantly longer to compute than each feature in S_j.[22] The portfolio can start by computing the easiest features, and iteratively compute the next set only if the expected benefit exceeds the cost of computation. More precisely:

(1) For each set S_j learn a model $c(S_j)$ that estimates time required to compute it. This could simply be average time scaled by input size.
(2) Divide the training examples into two sets. Using the first set, train models $M_1^i \cdots M_l^i$, with M_k^i predicting algorithm i's runtime using features in $\bigcup_{j=1}^{k} S_j$. Note that M_l^i is the same as the model for algorithm i in our basic portfolio methodology. Let M_k be a portfolio that selects $\arg\min_i M_k^i$.
(3) Using the second training set, learn models $D_1 \ldots D_{l-1}$, with D_k predicting the difference in runtime between the algorithms selected by M_k and M_{k+1} based

[22] We assume here that features' runtimes will have low variance across instances; this assumption holds in our WDP case study. If feature runtime variance makes it difficult to partition the features into time complexity sets, smart feature computation becomes somewhat more complicated.

on S_k. The second set should be used to avoid training the difference models on data to which the runtime models were fit.

Given an instance x, the portfolio now works as follows:

(4) For $j = 1$ to l
  (a) Compute features in S_j
  (b) If $D_j[x] > c(S_{j+1})[x]$, continue.
  (c) Otherwise, return with the algorithm predicted to be fastest according to M_j.

6.3. BOOSTING STEP 2: INDUCING HARD DISTRIBUTIONS. It is widely recognized that the choice of test distribution is important for empirically-driven algorithm development. In the absence of general techniques for generating instances that are both realistic and hard, the development of new distributions has usually been performed manually. An excellent example of such work is Selman et al. [1996], which describes a method of generating SAT instances near the phase transition threshold, which have been observed to be hard for most SAT solvers. More recently, there has been a conscious effort in the SAT community to provide generators for hard instances. For example, Achlioptas et al. [2005] and Jia et al. [2007] hid prespecified solutions in random formulas that appeared to be hard. Jia et al. [2004] generated random formulae based on statistical physics spin-glass models, highlighting the connection between physical phenomena and phase transitions in SAT.

Once we have decided to solve the algorithm selection problem by selecting among existing algorithms using a portfolio approach, it makes sense to reexamine the way we design and evaluate algorithms. Since the purpose of designing a new algorithm is to reduce the time that it will take to solve problems, designers should aim to produce new algorithms that complement an existing portfolio rather than seeking to make it obsolete. Since it is natural to build a new algorithm by attempting to optimize its average runtime on a given distribution, it is desirable to construct a new distribution with the property that if algorithm a_1 attains higher average performance on this distribution than algorithm a_2, a_1 would make a greater contribution to the existing portfolio on the original distribution (measured again in terms of average runtime) as compared to a_2. Note that here we make the simplifying assumption that the portfolio is always able to select the correct algorithm perfectly; note as well that a_1's contribution to the portfolio will only be weakly greater in the case where a_1 and a_2 are both worse than the portfolio on every instance.

Consider a setting in which the distribution of instances upon which we would like to optimize our performance is expressed as a distribution over one or more otherwise unrelated instance generators, and where we vary these generators' parameters uniformly within some given ranges. (We have been exploring just such a setting in our case study.) We call the overall distribution D, and write $D = D_g \cdot D_{p_i}$, where D_g is a distribution over instance generators with different parameter spaces, and D_{p_i} is a distribution over the parameters of the chosen instance generator i. Given a portfolio, the greatest opportunity for improvement is on instances that are hard for that portfolio, common in D, or both. More precisely, the maximum amount of performance improvement that can be achieved in a region of problem space is proportional to the expected amount of time the current portfolio will spend solving instances in that region. We propose to construct a new distribution that samples the original distribution in proportion to these maximal opportunities for improvement. This is analogous to the principle from boosting that new classifiers should be trained on instances that are hard for the existing ensemble, in the proportion that they occur in the original training set. Let H_f be a model of portfolio runtime based on instance features, constructed as the minimum of the models that constitute the portfolio. By normalizing, we can reinterpret this model as a density function h_f. By the argument above, we should generate instances from the product of this distribution and our original distribution, D:

$$D \cdot h_f(x) = \frac{D(x)h_f(x)}{\int Dh_f}.$$

Unfortunately, it is problematic to sample from $D \cdot h_f$: D is likely to be nonanalytic (since it is implemented through instance generators), while h_f depends on features and so can only be evaluated after an instance has been created. One way to sample from $D \cdot h_f$ is *rejection sampling* (see, e.g., Doucet et al. [2001]): generate problems from D and keep them with probability proportional to h_f. The main problem with rejection sampling is that it can generate a very large number of samples before accepting one.

Rejection is less likely when another distribution is available to guide the sampling process toward hard instances, as occurs in *importance sampling*. Test distributions usually have some tunable parameters $\overrightarrow{p}$, and although the hardness of instances generated with the same parameter values can vary widely, $\overrightarrow{p}$ will often be somewhat predictive of hardness. First, we consider the special case where D is expressed using a single instance generator. In this case, we can generate instances from $D \cdot h_f$ in the following way:

(1) Create a new hardness model H_p, trained using only $\overrightarrow{p}$ as features, and normalize it so that it can be used as a pdf, h_p.

(2) Generate a large number of instances from $D \cdot h_p$. Observe that we *can* sample from this distribution: first we sample directly from h_p, which is possible because it is a polynomial; this gives us parameter values that we can pass to the generator.

(3) Construct a distribution over instances by assigning each instance s probability proportional to $\frac{H_f(s)}{h_p(s)}$, and select an instance by sampling from this distribution.[23]

Observe that if h_p turns out to be helpful, hard instances from $D \cdot h_f$ will be encountered quickly. Even when h_p directs the search *away* from hard instances, observe that we will still sample from the correct distribution because the weights are divided by $h_p(s)$; this is true as long as h_p is nonzero wherever h_f is nonzero.

When D consists of multiple, unrelated generators, it can be difficult to learn a single H_p. A good solution is to factor h_p as $h_g \cdot h_{p_i}$, where h_g is a hardness model using only the choice of instance generator as a feature, and h_{p_i} is a hardness model in instance generator i's parameter space. Likewise, instead of using a single feature-space hardness model H_f, we can train a separate model for each generator

[23] In true rejection sampling, Step (2) would generate a single instance that would be then accepted or rejected in Step (3). Our technique approximates this process, but has the advantages that it does not require us to normalize H_f and that it guarantees we will be able to output an instance after generating a constant number of samples.

$H_{f,i}$ and normalize each to a pdf $h_{f,i}$.[24] The goal is now to generate instances from the distribution $D_g \cdot D_{p_i} \cdot h_{f,i}$, which can be done as follows:

(1) For every instance generator i, create a hardness model H_{p_i} with features $\overrightarrow{p_i}$, and normalize it to create a pdf, h_{p_i}.

(2) Construct a distribution over instance generators h_g, where the probability of each generator i is proportional to the average hardness of instances generated by i.

(3) Generate a large number of instances from $(D_g \cdot h_g) \cdot (D_{p_i} \cdot h_{p_i})$

   (a) select a generator i by sampling from $D_g \cdot h_g$;
   (b) select parameters for the generator by sampling from $D_{p_i} \cdot h_{p_i}$;
   (c) run generator i with the chosen parameters to generate an instance.

(4) Construct a distribution over instances by assigning each instance s from generator i probability proportional to $\frac{H_{f,i}(s)}{h_g(s) \cdot h_{p_i}(s)}$, and select an instance by sampling from this distribution.

Finally, note that these techniques have applications to the generation of distributions with other desired properties. For example, they can be used to induce "realistic" distributions that better reflect the sorts of problems expected to arise in practice. This can be achieved when it is possible to construct a function that scores the realism of a generated instance based on features of the instance. Essentially, we can use this function in place of the hardness model in the procedures described above.

7. *Portfolios and Hard Distributions: WDP Case Study*

We applied the ideas presented in Section 6 to our WDP case study. Here we present the results of these experiments, showing that empirical hardness models can be used in the WDP domain to build effective algorithm portfolios and to induce harder test distributions.

7.1. Boosting Step 1: Building Algorithm Portfolios. We considered portfolio performance on two data sets: the first of our fixed size data sets (1000 nondominated bids, 256 goods), and our variable size data set. In Section 2.4, we described the collection of CPLEX data for these two data sets. We also collected runtime data for other algorithms on each of these problem instances. On the fixed size data set, we collected runtime data for GL and CASS (see Section 2.1 for their descriptions). For the variable-size data set[25] we only considered CPLEX 8.0 and CASS, since the first data set indicated that GL provided relatively little benefit to a portfolio. On this data set, we established a 12-hour captime for CASS, motivated by the observation that CASS was unlikely to solve an instance within any reasonable time if it did not solve it in 12 hours.

[24] In our experimental work (e.g., Figures 31–33) we simply use hardness models H_f trained on the whole data set rather than using models trained on individual distributions. Learning per-distribution models would likely improve our results even further.

[25] Due to the problem discussed in Footnote 13, we did not collect CASS runtimes on the CATS Paths distribution. Hence, we have dropped Paths instances from the variable-size data set in this section.

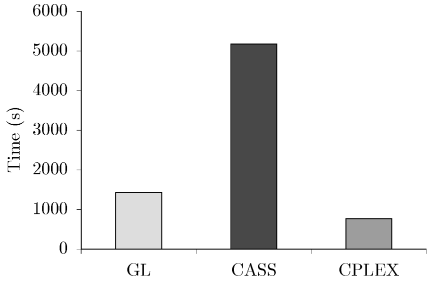

FIG. 22. Algorithm runtimes (test data, 1000 bids/256 goods).

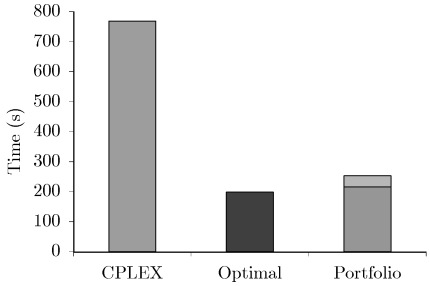

FIG. 23. Portfolio runtimes (test data, 1000 bids/256 goods).

We used the methodology from Section 2 to build regression models for GL and CASS in addition to the CPLEX models that we have already described. Figures 22 and 23 compare the (test set) average runtimes of our three algorithms (CPLEX, CASS, GL) to that of the portfolio on our fixed-size data set.[26] These averages were computed over instances on which *at least one* of the algorithms did not time out, and thus include some capped runtimes. Therefore, the bars in reality represent *lower bounds* on average runtimes on these data sets for constituent algorithms.

CPLEX would clearly be chosen under winner-take-all algorithm selection on both data sets. The "optimal" bar shows the performance of an ideal portfolio where algorithm selection is performed perfectly and with no overhead. The portfolio bar shows the time taken to compute features (light portion) and the time taken to run the selected algorithm (dark portion). Despite the fact that CASS and GL were much slower than CPLEX on average, the portfolio outperformed CPLEX by roughly a factor of 3. Moreover, neglecting the cost of computing features, our portfolio's selections on average took only 9% longer to run than the optimal selections.

Figures 24 and 25 show the frequency with which each algorithm would be selected in the ideal portfolio and was selected in our portfolio on the fixed-size data set. They illustrate the quality of our algorithm selection and the relative value of the three algorithms for the portfolio. It turns out that CASS was often significantly uncorrelated with CPLEX, and that most of the speedup in our portfolio came from choosing CASS on appropriate instances. Observe that the portfolio did not always make the right choice (in particular, it selected GL and CASS slightly more often than it should have). However, most of the mistakes made by our models

[26] Note the change of scale on the graph, and the repeated CPLEX bar.

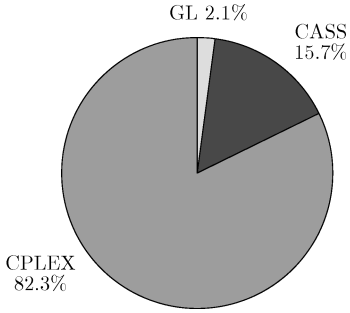

FIG. 24. Optimal selection (test data, 1000 bids/256 goods).

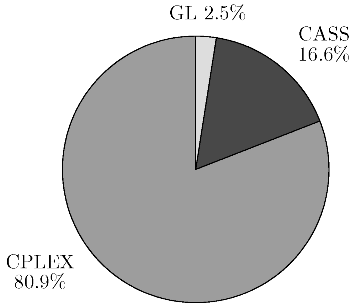

FIG. 25. Portfolio selection (test data, 1000 bids/256 goods).

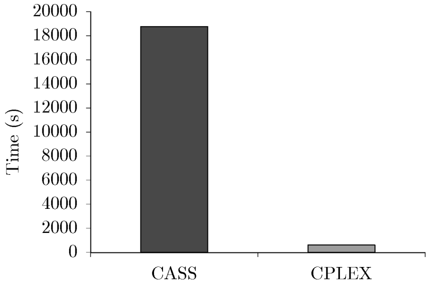

FIG. 26. Algorithm runtimes (variable size).

occurred when both algorithms had very similar running times. These mistakes were not very costly, explaining why our portfolio's choices had a running time so close to the optimal. This highlights an important point about our portfolio methodology: algorithm selection can be effective even with fairly weak empirical hardness models. It is easiest to discriminate among algorithms when their runtimes differ greatly, and when it becomes harder to discriminate between algorithms it also becomes less important to do so. (This comes back to the idea that classifiers for algorithm selection should be trained using a cost-sensitive error metric, discussed earlier in Section 6.2.1.3.)

Figures 26, 27, 28, and 29 describe portfolio performance on the variable-size data set. The average gain of using a portfolio was less dramatic here. This is because CPLEX was able to solve significantly harder instances than CASS, and thus the average runtime for the portfolio tracked CPLEX's runtime much more

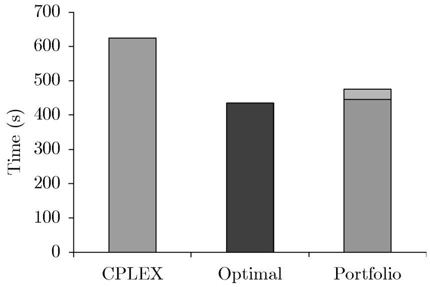

FIG. 27. Portfolio runtimes (variable size).

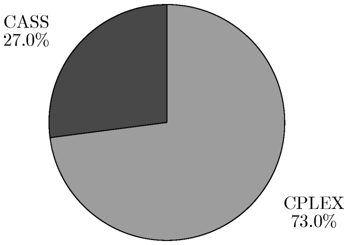

FIG. 28. Optimal selection (variable size).

closely than CASS's. As Figure 28 demonstrates, it is still the case that CASS was faster on roughly a quarter of the instances, and the portfolio often correctly selected CASS instead of CPLEX. However, the amount to be gained on this data set by choosing CASS over CPLEX was less dramatic. (We cannot be sure whether this was due to the different distribution of problem sizes or to changes in CPLEX in version 8.0; we suspect that it was a combination of both.) This data set illustrates that our portfolio methodology can still offer gains even when one algorithm is overwhelmingly stronger, though of course these gains are less impressive. It is noteworthy that the portfolio still *did* outperform CPLEX on this data set. Observe that the portfolio selected CASS more than a quarter of the time, and it could easily have achieved dramatically worse performance than CPLEX if even a small fraction of these picks had been instances on which CASS is very slow.

Some points previously discussed in Section 5 provide insight into reasons that an algorithm like CASS was able to provide such large gains over algorithms like CPLEX and GL on a significant fraction of instances. Unlike CASS, both GL and CPLEX use an LP relaxation heuristic. It is possible that when the number of constraints (and thus the node degree measures) increases, such heuristics become less accurate, or larger LP input size incurs substantially higher per-node costs. On the other hand, additional constraints reduce the size of the feasible search space. CASS often benefits when the search space becomes smaller; thus, CASS can achieve better overall performance on problems with a very large number of constraints.

These results show that our portfolio methodology can work very well even with a small number of algorithms, and when one algorithm's average performance is considerably better than the others'. Of course, it is likely that our techniques would have allowed us to construct a much stronger portfolio if we had used a larger set of algorithms and/or stronger competitors to CPLEX. As we mentioned earlier, we

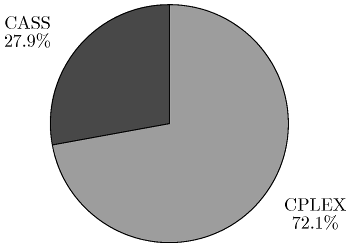

FIG. 29. Portfolio selection (variable size).

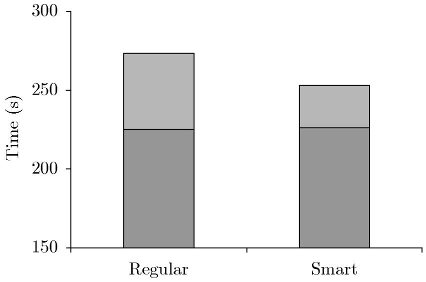

FIG. 30. Smart features (test data, 1000 bids/256 goods).

did just this for the SAT problem in our work on SATzilla [Nudelman et al. 2004a; Xu et al. 2007b, 2008].

7.1.1. *Portfolio Extensions.* Figure 30 shows the performance of the smart feature computation discussed in Section 6.2.4, with the upper part of the bar indicating the time spent computing features. (Observe that the y axis does not begin at 0.) Compared to computing all features, we reduce overhead by almost half with very slight degradation of portfolio performance.[27]

7.2. BOOSTING STEP 2: INDUCING HARD DISTRIBUTIONS. Figure 31 shows a runtime CDF for our original fixed-size distribution (i.e., our gross hardness graph from Figure 4 aggregated across generators) and the new, harder distribution induced using our sampling technique.[28] Both series in this figure represent data capped at the same limit. Because the original data was capped, there is no way to know if the hardest instances in the new distribution are harder than the hardest instances in the original distribution without rerunning experiments; note, however, that very few easy instances were generated. Instances in the induced distribution came predominantly from the CATS *arbitrary* distribution, with most of the rest from L3.

[27] This figure was obtained in our previous work [Leyton-Brown et al. 2002], using models that are slightly different from the models used for the rest of the results in this article. Unfortunately we cannot regenerate the figure for the same reason that we cannot add Paths data where it is missing: we no longer have access to the computers upon which we ran these experiments, and so we cannot collect data with which we could make accurate running time comparisons. However, we have verified that the performance of the models from our previous work [Leyton-Brown et al. 2002] is virtually identical to that of the new ones.

[28] Again, these results are based on models used in past work [Leyton-Brown et al. 2002].

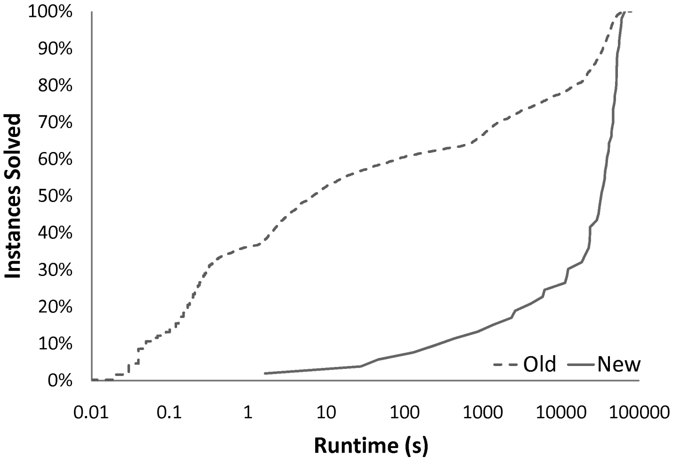

FIG. 31. Inducing harder distributions (test data, 1000 bids/256 goods).

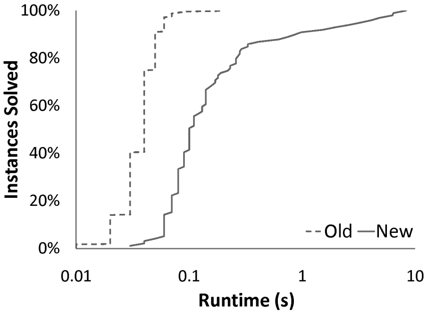

FIG. 32. Matching (test data, 1000 bids/256 goods).

Based on the wide spread of runtimes in our composite distribution D (seven orders of magnitude) and the high accuracy of our model h_f, one might argue that it was quite easy for our technique to generate harder instances. To demonstrate that our technique also works in more challenging settings, we sought an initial distribution with small runtime variance. As mentioned in Section 3.2.2, there has been ongoing discussion in the WDP literature about whether those CATS distributions that are relatively easy could be configured to be harder (see, e.g., Gonen and Lehmann [2000] and Sandholm et al. [2001]). We consider two distributions from CATS that, at their default settings, were easy with low variance: *matching* and *scheduling*. We show that these distributions can indeed be made much harder than originally proposed. Figures 32 and 33 show the CDFs of the runtimes of the ideal portfolio before and after our technique was applied. In fact, for these two distributions we generated instances that were (respectively) 100 and 50 times harder than anything we had previously seen! Moreover, the *average* runtime for the new distributions was greater than the observed *maximum* running time on the original distribution.

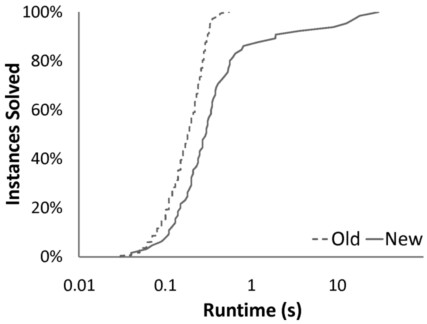

FIG. 33. Scheduling (test data, 1000 bids/256 goods).

8. *Conclusions*

In this article, we proposed a general methodology for understanding the empirical hardness of NP-complete problems on potentially complex, high-dimensional instance distributions. Although we have presented linear and quadratic regression as our main learning algorithms, our methods are more general and may be used with any learning technique. (We do, however, believe that regression techniques are more appropriate to this problem than classification techniques.) We believe that our methodology is applicable to a wide variety of hard computational problems.

Of course, a proposed empirical methodology is just an opinion without experimental evidence to support its use. To validate our methodology, we performed an extensive experimental investigation into the empirical hardness of the combinatorial auction winner determination problem. We identified structural, distribution-independent features of WDP instances and showed that—to our great initial surprise—they contain enough information to predict CPLEX's running time with high accuracy (see, e.g., Table II). Software for performing all of the feature selection, model building, subset selection and portfolio construction described in this paper is publicly available at `http://cs.ubc.ca/~kevinlb/downloads.html`, as is all the data needed to replicate our experiments.

We have also argued that, given accurate empirical hardness models, algorithm design should be guided by the boosting metaphor. Empirical hardness models can be used to combine algorithms together into a portfolio that outperforms each of its constituents. We argued that algorithm design should focus on problem instances upon which a portfolio of existing algorithms spends most of its time, and provided techniques for using empirical hardness models to induce such distributions automatically.

We performed experiments on WDP algorithms, and showed that a portfolio composed of CPLEX, CASS and GL can outperform CPLEX alone by a factor of 3—despite the fact that CASS and GL are *much* slower than CPLEX on average (see Figures 22 and 23). We were also able to induce a benchmark distribution that was much harder for our portfolio, and were even able to make specific CATS distributions much harder (see Figures 31, 32, and 33).

ACKNOWLEDGMENTS. The work presented in this article is based in part on previously published work [Leyton-Brown et al. 2002, 2003b, 2003a, 2006]. We would like to thank the following for their contributions to our work. First, our coauthors: Galen Andrew and Jim McFadden co-wrote the second and third of these articles with us; this article draws substantially on that work and in a few sections (e.g., 6.2.4 and 6.3) adapts text written by Galen and Jim. Holger Hoos and Alex Devkar were coauthors on our work on random SAT [Nudelman et al. 2004b], some ideas from which are reflected in the methodological sections of this article. The work described here has been further followed up on in articles co-written with Youssef Hamadi, Holger H. Hoos, Frank Hutter, and Lin Xu; please see the citations in Section 1.4. Second, we would like to acknowledge individuals who gave us assistance and helpful discussions: Rámon Béjar, Nando de Freitas, Carla Gomes, Henry Kautz, Ryan Porter, Bart Selman, Lyle Ungar and Ioannis Vetsikas. Thanks to Frank Hutter for identifying some last-minute bugs in our feature generation code, and especially to Lin Xu for fixing these bugs, rebuilding models using the corrected data, and helping with the generation of corrected tables and figures.

REFERENCES

ACHLIOPTAS, D. 2001. Lower bounds for random 3-SAT via differential equations. *Theoret. Comput. Sci. 265,* 1–2, 159–185.

ACHLIOPTAS, D., BEAME, P., AND MOLLOY, M. 2004. A sharp threshold in proof complexity yields lower bounds for satisfiability search. *J. Comput. Syst. Sci. 68,* 2, 238–268.

ACHLIOPTAS, D., GOMES, C. P., KAUTZ, H. A., AND SELMAN, B. 2000. Generating satisfiable problem instances. In *AAAI: Proceedings of the AAAI Conference on Artificial Intelligence*. 256–261.

ACHLIOPTAS, D., JIA, H., AND MOORE, C. 2005. Hiding satisfying assignments: Two are better than one. *J. Artif. Intell. Res. 24*, 623–639.

ANDERSON, A., TENHUNEN, M., AND YGGE, F. 2000. Integer programming for combinatorial auction winner determination. In *ICMAS: Proceedings of the International Conference on Multi-agent Systems*. 39–46.

BEAME, P., KARP, R., PITASSI, T., AND SAKS, M. 1998. On the complexity of unsatisfiability proofs for random k-CNF formulas. In *STOC: Proceedings of the Annual ACM Symposium on Theory of Computing*. ACM, New York, 561–571.

BREWER, E. 1994. Portable high-performance supercomputing: High-level platform-dependent optimization. Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA.

BREWER, E. 1995. High-level optimization via automated statistical modeling. In *SIGPLAN: Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*. ACM, New York, 80–91.

CARCHRAE, T., AND BECK, J. C. 2005. Applying machine learning to low knowledge control of optimization algorithms. *Computat. Intell. 21,* 4, 372–387.

CHALONER, K., AND VERDINELLI, I. 1995. Bayesian experimental design: A review. *Statist. Sci. 10*, 273–304.

CHEESEMAN, P., KANEFSKY, B., AND TAYLOR, W. M. 1991. Where the really hard problems are. In *IJCAI: Proceedings of the International Joint Conference on Artificial Intelligence*. 331–337.

COCCO, S., AND MONASSON, R. 2004. Heuristic average-case analysis of the backtrack resolution of random 3-satisfiability instances. *Theoret. Comput. Sci. 320,* 2-3, 345–372.

CRAMTON, P., SHOHAM, Y., AND STEINBERG, R. (EDS), 2006. *Combinatorial Auctions*. MIT Press, Cambridge.

DE FARIAS, D. P., AND MEGIDDO, N. 2004. How to combine expert (or novice) advice when actions impact the environment. In *NIPS: Proceedings of the Neural Information Processing Systems Conference*.

DE VRIES, S., AND VOHRA, R. 2003. Combinatorial auctions: A survey. *INFORMS J. Comput. 15,* 3, 284–309.

DIAO, Y., ESKESEN, F., PROEHLICH, S., HELLERSTEIN, J., SPAINHOWER, L., AND SURENDRA, M. 2003. Generic online optimization of multiple configuration parameters with application to a database server. *Proceedings of the IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*.

DOUCET, A., DE FREITAS, N., AND GORDEN, N. (EDS.) 2001. *Sequential Monte Carlo Methods in Practice*. Springer-Verlag.

DUBOIS, O., AND BOUFKHAD, Y. 1997. A general upper bound for the satisfiability threshold of random r-SAT formulae. *Journal of Algorithms 24,* 2, 395–420.

DUBOIS, O., BOUFKHAD, Y., AND MANDLER, J. 2000. Typical random 3-SAT formulae and the satisfiability threshold. In *SODA: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*. ACM, New York, 126–127.

EFRON, B., HASTIE, T., JOHNSTONE, I., AND TIBSHIRANI, R. 2004. Least angle regression. *Ann. Stat. 32*, 407–451.

EPSTEIN, S., FREUDER, E., WALLACE, R., MOROZOV, A., AND SAMUELS, B. 2002. The adaptive constraint engine. In *CP: Proceedings of the International Conference on Principles and Practice of Constraint Programming*. 525–540.

FRANCO, J. 2001. Results related to threshold phenomena research in satisfiability: Lower bounds. *Theoret. Comput. Sci. 265,* 1–2, 147–157.

FRIEDMAN, J. 1991. Multivariate adaptive regression splines. *Ann. Stat. 19,* 1, 1–141.

FRIEZE, A., AND SUEN, S. 1996. Analysis of two simple heuristics on a random instance of k-SAT. *J. Algor. 20,* 2, 312–355.

FUJISHIMA, Y., LEYTON-BROWN, K., AND SHOHAM, Y. 1999. Taming the computational complexity of combinatorial auctions: Optimal and approximate approaches. In *IJCAI: Proceedings of the International Joint Conference on Artificial Intelligence*. 548–553.

GAGLIOLO, M., AND SCHMIDHUBER, J. 2006. Learning dynamic algorithm portfolios. *Ann. Mathematics and Artificial Intelligence 47,* 3-4, 295–328.

GEBRUERS, C., AND GUERRI, A. 2004. Machine learning for portfolio selection using structure at the instance level. In *Principles and Practice of Constraint Programming, Doctoral Consortium*.

GEBRUERS, C., GUERRI, A., HNICH, B., AND MILANO, M. 2004. Making choices using structure at the instance level within a case based reasoning framework. In *CPAIOR: Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. 380–386.

GEBRUERS, C., HNICH, B., BRIDGE, D., AND FREUDER, E. 2005. Using CBR to select solution strategies in constraint programming. In *ICCBR: Proceedings of the International Conference on Case-Based Reasoning*. 222–236.

GOLDSMITH, S., AIKEN, A., AND WILKERSON, D. 2007. Measuring empirical computational complexity. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, New York, 395–404.

GOLDSZMIDT, M. 2007. Making life better one large system at a time: Challenges for UAI research. In *UAI: Proceedings of the Conference on Uncertainty in Artificial Intelligence*.

GOMES, C., FERNÁNDEZ, C., SELMAN, B., AND BESSIÈRE, C. 2004. Statistical regimes across constrainedness regions. In *CP: Proceedings of the International Conference on Principles and Practice of Constraint Programming*.

GOMES, C., AND SELMAN, B. 2001. Algorithm portfolios. *Artif. Intell. 126,* 1-2, 43–62.

GOMES, C., SELMAN, B., CRATO, N., AND KAUTZ, H. 2000. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Automat. Reason. 24,* 1–2, 67–100.

GOMES, C. P., AND SELMAN, B. 1997. Problem structure in the presence of perturbations. In *AAAI: Proceedings of the AAAI Conference on Artif. Intell.* 221–226.

GONEN, R., AND LEHMANN, D. 2000. Optimal solutions for multi-unit combinatorial auctions: Branch and bound heuristics. In *EC: Proceedings of the ACM Conference on Electronic Commerce*. ACM, New York, 13–20.

GONEN, R., AND LEHMANN, D. 2001. Linear programming helps solving large multi-unit combinatorial auctions. Tech. Rep. TR-2001-8, Leibniz Center for Research in Computer Science. April.

GUO, H., AND HSU, W. 2007. A machine learning approach to algorithm selection for NP-hard optimization problems. *Ann. Oper. Res. 156,* 1 (Dec.), 61–82.

HASTIE, T., TIBSHIRANI, R., AND FRIEDMAN, J. 2001. *Elements of Statistical Learning*. Springer-Verlag, Berlin, Germany.

HOOS, H., AND BOUTILIER, C. 2000. Solving combinatorial auctions using stochastic local search. In *AAAI: Proceedings of the AAAI Conference on Artif. Intell.* 22–29.

HOOS, H. H., AND STÜTZLE, T. 1999. Towards a characterisation of the behaviour of stochastic local search algorithms for SAT. *Artif. Intell. 112,* 1-2, 213–232.

HOOS, H. H., AND STÜTZLE, T. 2004. *Stochastic Local Search: Foundations and Applications*. Morgan-Kaufmann, Reading.

HORVITZ, E., RUAN, Y., GOMES, C., KAUTZ, H., SELMAN, B., AND CHICKERING, M. 2001. A Bayesian approach to tackling hard computational problems. In *UAI: Proceedings of the Conference on Uncertainty in Artif. Intell.* 235–244.

HUBERMAN, B., LUKOSE, R., AND HOGG, T. 1997. An economics approach to hard computational problems. *Science 265*, 51–54.

HUTTER, F., HAMADI, Y., HOOS, H. H., AND LEYTON-BROWN, K. 2006. Performance prediction and automated tuning of randomized and parametric algorithms. In *CP: Proceedings of the International Conference on Principles and Practice of Constraint Programming*. Lecture Notes in Computer Science, vol. 4204. Springer-Verlag, Berlin, Germany, 213–228.

JIA, H., MOORE, C., AND SELMAN, B. 2004. From spin glasses to hard satisfiable formulas. In *SAT: Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*. 199–210.

JIA, H., MOORE, C., AND STRAIN, D. 2007. Generating hard satisfiable formulas by hiding solutions deceptively. *J. Artif. Intell. Res. 28*, 107–118.

KNUTH, D. 1975. Estimating the efficiency of backtrack programs. *Math. Comput. 29,* 129, 121–136.

KOHAVI, R., AND JOHN, G. 1997. Wrappers for feature subset selection. *Artif. Intell. 97,* 1–2, 273–324.

KOLAITIS, P. 2003. Constraint satisfaction, databases and logic. In *IJCAI: Proceedings of the International Joint Conference on Artificial Intelligence*. 1587–1595.

KORF, R., AND REID, M. 1998. Complexity analysis of admissible heuristic search. *AAAI: Proceedings of the AAAI Conference on Artificial Intelligence*, 305–310.

LAGOUDAKIS, M., AND LITTMAN, M. 2000. Algorithm selection using reinforcement learning. In *ICML: Proceedings of the International Conference on Machine Learning*. 511–518.

LAGOUDAKIS, M., AND LITTMAN, M. 2001. Learning to select branching rules in the DPLL procedure for satisfiability. In *SAT: Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*. 344–359.

LEYTON-BROWN, K., NUDELMAN, E., ANDREW, G., MCFADDEN, J., AND SHOHAM, Y. 2003a. Boosting as a metaphor for algorithm design. In *CP: Proceedings of the International Conference on Principles and Practice of Constraint Programming*. 899–903.

LEYTON-BROWN, K., NUDELMAN, E., ANDREW, G., MCFADDEN, J., AND SHOHAM, Y. 2003b. A portfolio approach to algorithm selection. In *IJCAI: Proceedings of the International Joint Conference on Artificial Intelligence*. 1542–1543.

LEYTON-BROWN, K., NUDELMAN, E., AND SHOHAM, Y. 2002. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *CP: Proceedings of the International Conference on Principles and Practice of Constraint Programming*. 556–572.

LEYTON-BROWN, K., NUDELMAN, E., AND SHOHAM, Y. 2006. Empirical hardness models for combinatorial auctions. In *Combinatorial Auctions*. MIT Press, Cambridge, Chap. 19, 479–504.

LEYTON-BROWN, K., PEARSON, M., AND SHOHAM, Y. 2000a. Towards a universal test suite for combinatorial auction algorithms. In *EC: Proceedings of the ACM Conference on Electronic Commerce*. ACM, New York, 66–76.

LEYTON-BROWN, K., SHOHAM, Y., AND TENNENHOLTZ, M. 2000b. An algorithm for multi-unit combinatorial auctions. In *AAAI: Proceedings of the AAAI Conference on Artificial Intelligence*. 56–61.

LOBJOIS, L., AND LEMAÎTRE, M. 1998. Branch and bound algorithm selection by performance prediction. In *AAAI: Proceedings of the AAAI Conference on Artificial Intelligence*. 353–358.

MASON, R. L., GUNST, R. F., AND HESS, J. L. 2003. *Statistical Design and Analysis of Experiments*. Wiley-Interscience, New York.

MONASSON, R., ZECCHINA, R., KIRKPATRICK, S., SELMAN, B., AND TROYANSKY, L. 1998. Determining computational complexity for characteristic 'phase transitions'. *Nature 400*, 133–137.

NISAN, N. 2000. Bidding and allocation in combinatorial auctions. In *EC: Proceedings of the ACM Conference on Electronic Commerce*. ACM, New York, 1–12.

NUDELMAN, E., LEYTON-BROWN, K., DEVKAR, A., SHOHAM, Y., AND HOOS, H. H. 2004a. SATzilla: An algorithm portfolio for SAT. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing, SAT 2004 Competition: Solver Descriptions*. 13–14.

NUDELMAN, E., LEYTON-BROWN, K., DEVKAR, A., SHOHAM, Y., AND HOOS, H. H. 2004b. Understanding random SAT: Beyond the clauses-to-variables ratio. In *CP: Proceedings of the International Conference on Principles and Practice of Constraint Programming*. 438–452.

RICE, J. R. 1976. The algorithm selection problem. *Adv. Comput. 15*, 65–118.

ROTHKOPF, M., PEKEČ, A., AND HARSTAD, R. 1998. Computationally manageable combinatorial auctions. *Manage. Sci. 44,* 8, 1131–1147.

RUAN, Y., HORVITZ, E., AND KAUTZ, H. 2002. Restart policies with dependence among runs: A dynamic programming approach. In *CP: Proceedings of the International Conference on Principles and Practice of Constraint Programming*. 573–586.

SANDHOLM, T. 1999. An algorithm for optimal winner determination in combinatorial auctions. In *IJCAI: Proceedings of the International Joint Conference on Artificial Intelligence*. 542–547.

SANDHOLM, T. 2002. Algorithm for optimal winner determination in combinatorial auctions. *Artif. Intell. 135,* 1, 1–54.

SANDHOLM, T., SURI, S., GILPIN, A., AND LEVINE, D. 2001. CABOB: A fast optimal algorithm for combinatorial auctions. In *IJCAI: Proceedings of the International Joint Conference on Artificial Intelligence*. 1102–1108.

SANDHOLM, T., SURI, S., GILPIN, A., AND LEVINE, D. 2005. CABOB: A fast optimal algorithm for winner determination in combinatorial auctions. *Manage. Sci. 51,* 3, 374–390.

SCHAPIRE, R. 1990. The strength of weak learnability. *Mach. Learn. 5*, 197–227.

SCHMEE, J., AND HAHN, G. J. 1979. A simple method for regression analysis with censored data. *Technometrics 21,* 4, 417–432.

SELMAN, B., MITCHELL, D. G., AND LEVESQUE, H. J. 1996. Generating hard satisfiability problems. *Artif. Intell. 81,* 1-2, 17–29.

SLANEY, J., AND WALSH, T. 2001. Backbones in optimization and approximation. In *IJCAI: Proceedings of the International Joint Conference on Artificial Intelligence*. 254–259.

STREETER, M., GOLOVIN, D., AND SMITH, S. F. 2007. Combining multiple heuristics online. In *AAAI: Proceedings of the AAAI Conference on Artificial Intelligence*. 1197–1203.

WILLIAMS, R., GOMES, C., AND SELMAN, B. 2003. Backdoors to typical case complexity. In *IJCAI: Proceedings of the International Joint Conference on Artificial Intelligence*. 1173–1178.

XU, L., HOOS, H. H., AND LEYTON-BROWN, K. 2007a. Hierarchical hardness models for SAT. In *CP: Proceedings of the International Conference on Principles and Practice of Constraint Programming*. Lecture Notes in Computer Science, vol. 4741. Springer-Verlag, Berlin, Germany, 696–711.

XU, L., HUTTER, F., HOOS, H. H., AND LEYTON-BROWN, K. 2007b. SATzilla-07: The design and analysis of an algorithm portfolio for SAT. In *CP: Proceedings of the International Conference on Principles and Practice of Constraint Programming*. Lecture Notes in Computer Science, vol. 4741. Springer-Verlag, Berlin, Germany, 712–727.

XU, L., HUTTER, F., HOOS, H. H., AND LEYTON-BROWN, K. 2008. SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res. 32*, 565–606.

ZHANG, W. 1999. *State-Space Search: Algorithms, Complexity, Extensions, and Applications*. Springer-Verlag, Berlin, Germany.

ZHANG, W. 2001. Phase transitions and backbones of 3-SAT and maximum 3-SAT. In *CP: Proceedings of the International Conference on Principles and Practice of Constraint Programming*.

ZHENG, A., JORDAN, M., LIBLIT, B., AND AIKEN, A. 2003. Statistical debugging of sampled programs. In *NIPS: Proceedings of the Neural Information Processing Systems Conference*.

RECEIVED OCTOBER 2005; REVISED OCTOBER 2008; ACCEPTED MARCH 2009